UNIVERSIDAD COMPLUTENSE DE MADRID

FACULTAD DE INFORMÁTICA
Departamento de Arquitectura de Computadores y Automática

TESIS DOCTORAL

Desarrollo de solvers CFD híbridos en plataformas heterogéneas

MEMORIA PARA OPTAR AL GRADO DE DOCTOR

PRESENTADA POR

Pedro Valera Lara

Directores

Manuel Prieto Matías
Alfredo Pinelli

Madrid, 2016

Desarrollo de solvers CFD híbridos en plataformas heterogéneas

Departamento de Arquitectura de Computadores y Automática

Universidad Complutense de Madrid

TESIS DOCTORAL

Pedro Valero Lara

Madrid, 2015

Desarrollo de solvers CFD híbridos en plataformas heterogéneas

Departamento de Arquitectura de Computadores y Automática

Universidad Complutense de Madrid

TESIS DOCTORAL

Pedro Valero Lara

Directores:
Dr. Manuel Prieto Matías
Dr. Alfredo Pinelli

Madrid, 2015

Desarrollo de solvers CFD híbridos en plataformas heterogéneas

Memoria presentada por Pedro Valero Lara para optar al grado de Doctor por la Universidad Complutense de Madrid, en el programa ingeniería informática.

Development of hybrid CFD solvers on heterogeneous platforms

Disertation submitted by Pedro Valero Lara to the Complutense University of Madrid in partial fulfilment of the requirements for the degree of doctor of computer science and engineering.

Directores/Advisors: Manuel Prieto Matías
Alfredo Pinelli

Madrid, 2015.

A scientist must be free to raise any question, to doubt any assertion, to correct mistakes.

Robert Oppenheimer

Why this magnificent scientific technology, which saves work and makes life easier, bring us so little happiness? The answer is simply, because we have not learned to use it wisely.

Albert Einstein

Agradecimientos

La finalización de cualquier tesis requiere mucho esfuerzo y dedicación a lo largo de un extenso periodo de tiempo. La soledad ha formado parte de ese tiempo, durante el cual se han afrontado, de la mejor manera que he sabido, nuevos problemas y retos. Gracias a estos retos, uno mismo sale más fortalezido a nivel personal y profesional.

En este trabajo han participado muchas personas, en especial mis directores Dr. Manuel Prieto Matías y Dr. Alfredo Pinelli, para los cuales sólo tengo palabras de agradecimiento y admiración. Me gustaría agradecerles, todo el esfuerzo y dedicación ofrecido durante todo este tiempo.

También agradecer al Centro de Invesitagiones Energéticas, Medio Ambientales y Técnologicas (CIEMAT), y en particular a mi director Dr. Alfredo Pinelli, quien fue director de la Unidad de Modelización de Procesos, todos los recursos disponibles a mi alcance, junto al Centro Extremeño de Tecnologías Avanzadas (CETA-CIEMAT). En particular, a todos los integrantes de la Unidad de Modelización de Procesos, lugar donde realice la mayor parte de este trabajo.

Igualmente agradecer a la Universidad Complutense de Madrid, y en particular a mi director Dr. Manuel Prieto Matías, y a los demás miembros del Departamento de Arquitectura de Computadores y Automática, por todo el apoyo recibido.

Por último, mis agradecimientos al Dr. Alistair Rewell por abrirme la oportunidad de estar un tiempo en la University of Manchester, en particular en el School of Mechanical, Aerospace, and Civil Engineering, en donde pude colaborar con otros investigadores sobre temas relacionados con este trabajo, y tener acceso a otros recursos.

Pedro Valero Lara, 2015

Resumen

La comunidad de dinámica de fluidos computacional (CFD) siempre ha explorado nuevas formas para aprovechar las plataformas de computación de alto rendimiento en su continua búsqueda de simulaciones más rápidas y precisas. Durante los últimos años, la irrupción de las arquitecturas heterogéneas ha sido una de las tendencias más importantes en este campo y se han creado nuevos desafíos y oportunidades para optimizar el rendimiento de los *solvers* considerados estado del arte. En este trabajo hemos explorado algunas de estas nuevas oportunidades.

Nuestros *solvers* objetivo se enmarcan en el dominio de los flujos incompresibles. A pesar de los significativos avances que han logrado las metodologías más avanzadas en este campo, un aspecto que sigue necesitando de más investigación es la aceleración de los *solvers* incompresibles, en particular cuando se manejan problemas de gran dimensión con geometrías complejas. El coste computacional de este tipo de *solvers* está dominado frecuentemente por la solución de la Ecuación de Poisson para la determinación de la presión. Nuestra primera contribución en esta tesis ha explorado la aceleración en sistemas heterogéneos de los denominados métodos rápidos para la solución de esta ecuación. Primero investigamos el rendimiento de diferentes algoritmos en procesadores multicore y en GPUs por separado y posteriormente estudiamos la ejecución conjunta en ambos tipos de procesadores. Como era previsible, en los procesadores multicore las aproximaciones de grano grueso proporcionan los mejores resultados, mientras que en GPUs es mejor utilizar alternativas de grano fino basadas en estrategias de reducción cíclica extendidas. Nuestra principal contribución de esta parte de la tesis ha sido el diseño de una aproximación heterogénea que es capaz de combinar ambas estrategias y beneficiarse del solapamiento entre CPUs y GPUs. Nuestro diseño estuvo inspirado en la implementación 2D, donde el paralelismo es limitado e introduce importantes penalizaciones en la implementación GPU homogénea. Sin embargo, hemos encontrado que el uso combinado de CPU y GPU simultaneamente, también proporciona los mejores resultados para problemas 3D, a pesar de la mayor intensidad aritmética y paralelismo que aparece en este caso.

Desafortunadamente, el rendimiento global que estos *solvers* rápidos no satisface los objetivos que nos habíamos establecido. Es por ello que en la segunda parte

de esta tesis hemos tratado de superar esa limitación intrínseca de los *solvers* basados en Navier Stokes estudiando como alternativa el método de Lattice-Boltzmann (LBM). El diseño de implementaciones paralelas de LBM se ha estudiado de forma extensiva y algunos trabajos recientes han mostrado que puede alcanzar grandes rendimientos en aceleradores del tipo GPU. Sin embargo, las simulaciones que pretendemos realizar presentan geometrías complejas y no basta con *solvers* LBM puros. Una aproximación prometedora para tratar estos problemas es la combinación de LBM con el método de las fronteras immersas (IB: *Immersed Boundaries*). En primer lugar hemos comprobado como implementaciones directas de ambos métodos (LBM e IB) por separado no escalan bien, ya que la corrección IB degrada de forma notable el rendimiento global. Nuestra principal contribución ha sido el diseño de una implementación híbrida que permite solapar la ejecución de ambos métodos (LBM e IB) en plataformas heterogéneas, ocultando de forma efectiva las penalizaciones introducidas por IB. De hecho, para escenarios físicos de interés, con fracciones solido volumen realistas, el *solvers* híbrido propuesto es capaz de ocultar totalmente la penalización causada por la corrección IB. De este modo, alcanzamos globalmente en simulaciones complejas el rendimiento de los *solvers* LBM puros.

Notablemente, y esta es una de las conclusiones finales de este trabajo, hemos usado el mismo patrón para acelerar de forma efectiva los diferentes *frameworks* CFD explorados en esta tesis. La principal idea de este patrón ha consistido en la restructuración de los códigos para permitir una mejor coordinación entre el procesador *host* y el acelerador, permitiendo el solapamiento de sus ejecuciones. Con dicho solapamiento hemos sido capaces de ocular (1) el coste de las transferencias de datos entre el *host* y el acelerador y (2) las penalización causada por cuellos de botella secuenciales, es decir, hemos usado el procesador *host* como una acelerador de fases secuenciales del código.

Summary

The Computational Fluid Dynamic (CFD) community has always explored new ways to leverage high-performance computing platforms in its never-ending quest for faster and more accurate simulations. Over the last few years, the emergence of heterogeneous architectures has been one of the most important trends in this field and it has created new challenges and opportunities for performance optimization in state-of-the-art solvers. In this work, we have explored some of these new opportunities.

Our target solvers are in the incompressible flow domain. Despite the significant advances that have been achieved by state-of-the-art methodologies, one aspect that still deserves further investigation is the acceleration of incompressible solvers, particularly when dealing with large problems with complex geometries. The computational cost of these kind of solvers are usually dominated by the solution of the Poisson equation for determining pressure. Our first contribution in this thesis has explored the acceleration on heterogeneous systems of the well-known Fast Poisson solvers. First, we investigated the performance of different algorithms on multicore and GPU processors and afterwards we consider the execution on both processors simultaneously. As expected, on multicore processors coarse grain approaches give the best performance, whereas on GPUs it is better to use fine grain alternatives based on cyclic reduction. Our main contribution of this part of the thesis has been the design of a heterogeneous approach that is able to combine both strategies and benefit from CPU-GPU overlapping. Our design was inspired by the overheads that the 2D homogeneous GPU implementation suffers due to limited parallelism. However, we have found that it is also the best choice for the 3D case, despite of the higher arithmetic intensity and parallelism of those problems.

Unfortunately, the overall performance of those parallel fast solvers do not meet the goals envisioned. This is why, in the second part of this thesis, we have tried to overcome the intrinsic limitation of Navier-Stokes solvers studying as an alternative the Lattice-Boltzmann method (LBM). The design and implementation of parallel LBM solvers have been extensively studied and several recent works have shown it can achieve impressive performance on data-parallel accelerators. However, pure LBM solvers are not enough for our target simulations with complex geometries. A

promising approach in this direction is the combination of LBM with the Immersed Boundary (IB) method. First, we have found that a straightforward implementation of both methods by processing each method as an independent component, does not scale well since the IB correction degrades the overall performance. Our main contribution has been the design of hybrid LBM-IB implementation that effectively hide such overheads thanks to the overlapping of both components on the heterogeneous platform, with excellent performance results. In fact, for interesting physical scenarios, with realistic solid volume fractions, the proposed hybrid solvers are able to hide the overheads caused by the IB correction. Overall, we match the performance of state-of-the-art pure LBM solvers on more complex simulations.

Notably, and this is one of the overall conclusion behind this work, we have used the same computing pattern to effectively accelerate the different frameworks explored in this thesis. The main idea behind such pattern is to restructure the different codes to allow a better coordination between the host processor and the accelerator, allowing us the overlapping of their executions. With such overlapping we have been able to hide (1) the cost of data transfers between the host and the accelerator and (2) the overheads caused by sequential bottlenecks, i.e. the host processors is used as an accelerator of the sequential phases found in the different codes.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In this Chapter we give an introduction to Computational Fluid Dynamics (CFD) and Heterogeneous Computing, together with a general discussion which frames this work and the main publications arising from it. The aim is to set out the context of this work, summarize our objectives and main findings and establish the significance of this work.

The CFD community has always explored new ways to leverage emerging computing platforms in its never-ending quest for faster and more accurate simulations. The widespread usage of multi-core and heterogeneous architectures, which we have experienced over the last few years, has created new challenges and opportunities for performance optimization in advanced CFD solvers. In this work we have tried to address some of these challenges and opportunities.

In Section 1.1, we first introduce some of the basic ideas behind CFD and present an historical perspective (see [11, 10] for a complete review). We then summarize the main goals addressed in this thesis, and in the following subsections we discuss in some detail the three specific topics (solvers) explored in this work,

including a review of the related work.

As mentioned above, our main focus has been on mapping those solvers into emerging heterogeneous platforms found in current High Performance Computing (HPC) systems. While the computing community is building tools and libraries to ease the use of such systems, its effective use still requires a deep knowledge of low-level programming and a good understanding of the underlying hardware. Thus, in Section 1.2, we describe the computing platforms used in this work, and in Section 1.3, we list the main publications related to this thesis.

Finally, in Section 1.4 we summarize the main conclusions of this work.

1.1. Computational fluid dynamics

1.1.1. Basic philosophy of CFD

CFD is a set of numerical methods applied to obtain approximate solutions of problems of fluid dynamics and heat transfer [170]

According to this definition, CFD is not a science by itself, but a way to apply the methods of one discipline (numerical analysis) to another (heat and mass transfer) [170]. The physics of any fluid flow is governed by the fundamental principles, which can be expressed in terms of equations [11]. CFD is, in part, the art of replacing the governing equations of the fluid flow with a discrete series of numbers, and advancing these expressions in space and time to achieve, numerically, a description of the flow field. Some problems allow the immediate solution of the flow field without advancing in time or space, and others involve integral equations or statistical distributions rather than partial differential equations [11]. In any case,

all problems involve the manipulation, and the solution of discrete systems. The final result of CFD is indeed a collection of numbers, in contrast to an analytical solution. However, in the long run the objective of most engineering analyses is a quantitative description of the problem, i.e., numbers [11].

Obviously, high performance computing (HPC) is the instrument which has allowed the advance of fluid simulations [11]. CFD solutions require the repetitive management of millions of numbers. Therefore, the advances and its applications to problems of increasing detail and sophistication are intimately related to advances in computer hardware, particularly in regard to storage and execution speed [11]. This is why one of the strongest forces driving the development of HPC during the 80's and the 90's came from the CFD community [54]. Indeed, CFD continues to be one of the major drivers for advances in HPC [89], although in recent years other fields such as Big Data and Analytics are attracting more attention.

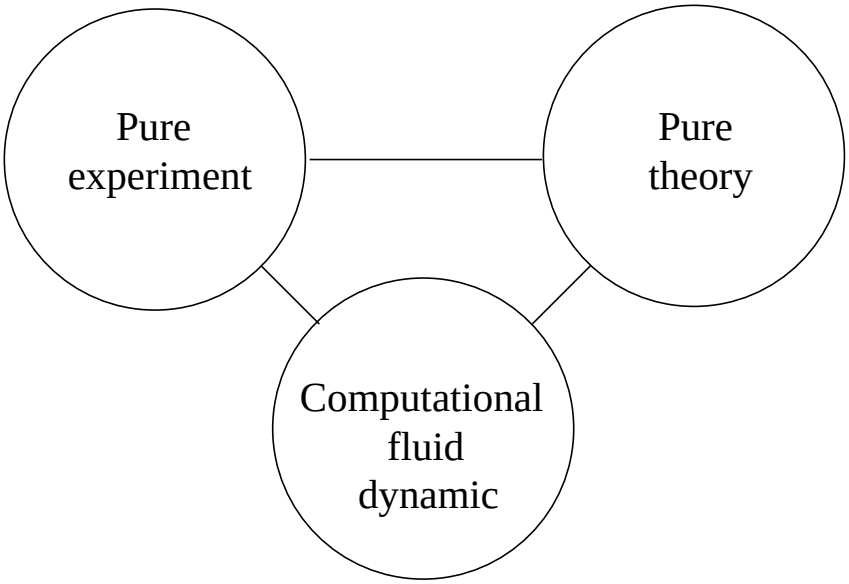


Figure 1.1: The "three dimensions" of fluid dynamics [159].

Throughout most of the twentieth century the study and practice of fluid dynamics (indeed, all of physical science and engineering) involved the use of pure theory and pure experiment [11]. However, CFD has become so important that nowadays it can be viewed as a new third dimension in fluid dynamics [159]. This relationship is graphically shown in Figure 1.1. From 1687, with the publication of Issac Newton's Principia, to the mid-1960's, fluid mechanics advanced through a combination of experiments and theoretical analysis, which always required the use of simplified models to obtain solutions of the governing equations [159]. These solutions have the distinct advantage of immediately identifying some of the fundamental parameters of a given problem, and explicitly demonstrating how the problems are influenced by the variation of these parameters. On the other hand, they present a disadvantage of not including all the required physics of the flow. The advent of computing in the 60's allowed CFD to mitigate these problems [159]. With its ability to handle the governing equations in "full" form, and to include detailed physical phenomena such as chemical reactions, CFD became a popular tool in engineering analyses. Now, CFD supports and complements both pure experiment and pure theory [11]. CFD and supercomputers will remain a third dimension in fluid dynamics, of equal importance to experiment and theory. They have taken a permanent place in all aspects of fluid dynamics, from basic research to engineering design [159].

1.1.2. Historical perspective. CFD as a research tool.

Perhaps, the first major example of CFD was the work of Kopal [83], who in 1947 compiled massive tables of supersonic flow over sharp cones by numerically

solving the governing differential equations. The computing was carried out on a primitive digital computer at Massachusetts Institute of Technology [159]. The first generation of CFD solutions appeared during the 1950's and early 1960's, spurred by the simultaneous advent of computers and the need to solve the high velocity, high temperature re-entry body problem [159]. High temperatures necessitated the inclusion of vibrational energies and chemical reactions in flow problems. Such physical phenomena generally cannot be solved analytically, even for simple geometries. Therefore, numerical solutions of the governing equations on computer systems became absolute necessary. Example of these first generation computations are the pioneering work of Fay [42] and Blottner [19, 20], for boundary layers, and Hall et al. [60] for inviscid flows [159].

In 1970, the existing computers and algorithms restricted all practical solutions basically to two-dimensional flows. The real world of fluid is mainly a three-dimensional world. The storage and speed capacity of computer at that time were not sufficient to manage three-dimensional practical fashion. Nevertheless, the story changed drastically in 1990. Today, three-dimensional solvers are abundant, although it is necessary a great deal of human and computer resources to successfully carry out such applications. They are increasing in importance within industry and government facilities [11].

Modern CFD complemets the use of wind tunnel testing and pure experiments, to study and validate physics problems [11]. This is related to the rapid decrease in the cost of computations compared to the cost of real experiments. As a result, the calculation of the physics characteristics via application of CFD is becoming economically cheaper than measuring the same characteristics by other means. CFD offers the opportunity to obtain detailed flow field information, some of which is ei-

ther difficult or impossible to be measured. Overall, inherent in the above discussion is the assumption that CFD results are accurate as well as cost effective [11, 10].

It is important to highlight that the results of CFD are only as valid as the physical models incorporated in the governing equations and boundary conditions, and therefore are subject to error, especially for complex experiments [11, 10]. Additionally, truncation error associated with the particular algorithm used to obtain a numerical solution, as well as round-off errors, both combine to compromise the accuracy of CFD results. Despite these drawbacks, the results of CFD are amazingly accurate for a very large number of applications [11, 10]. Indeed, it is possible to find a large number of problems which can be adequately handle by CFD: Supersonic flows, turbulent flows, combustion, solid-fluid interaction, blood flows are just a few examples. In many areas of applications, the basic methodologies are well established and have been implemented into commercial software packages. Nevertheless, it is still necessary to develop advance CFD methods for complex flow problems.

Every method can be divided into two subgroups, implicit-explicit [11, 10] and compressible-incompressible [11, 10, 45, 25]. Let us consider the following model equation so that we may explain the differences among both subgroups. We assume that the dependent variable u is a function of x (space) and t (time).

$$\frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial^2 x^2} \tag{1.1}$$

This simple equation is chosen for convenience. It can be discretized obtaining a first order in time (n) and second order in space (i) equation by replacing the time derivative with a forward difference and the spatial derivative with a central

difference [11, 170, 10, 25].

$$\frac{u_i^{n+1} - u_i^n}{\Delta t} = \frac{u_{i+1}^n - 2u_i^n + u_{i-1}^n}{\Delta x^2} \tag{1.2}$$

Examining the above equation, we see that it only contains one unknown, u_i^{n+1}. The dependent variables at time $n+1$ can be obtained *explicitly* from the known results at time n. On the other hand, if we apply the Crank-Nicolson discretization, one obtains:

$$\frac{u_i^{n+1} - u_i^n}{\Delta t} = \frac{u_{i+1}^{n+1} + u_{i+1}^n - 2u_i^{n+1} - 2u_i^n + u_{i-1}^{n+1} + u_{i-1}^n}{\Delta x^2} \tag{1.3}$$

Here the unknowns u^{n+1} are not only expressed in terms of the knowns u^n, but also in terms of dependent variables at time $n+1$ and this is an example of an implicit solution.

The explicit approaches are relatively much simpler to set up. However in many cases Δt must be very small to achieve enough stability which can lead to long running times. Otherwise the implicit approach can maintain the stability over much larger values of Δt. Nevertheless this approach present greater complexities to set up as massive matrix manipulations are necessary at each time step [11, 10].

Compressible flows are those that suffer changes in fluid density, for instance gases [11, 45, 25]. The study of these flows presents an important relevance to jet engines, gas pipelines, high-speed aircrafts and many other fields. Otherwise in incompressible flows [11] the density is maintain constant within a fluid domain that moves with the velocity of the fluid. In other words, the volume is constant for a fluid element in incompressible flows. The flow of a liquid, such as water, can be considered to be incompressible obtaining a high degree of accuracy. All methods

considered in this dissertation use the explicit-incompressible approach.

Today, the impact of CFD in research has consolidated. It consists of a set of different methodologies [11, 10, 170] such as Finite Difference Method (FDM) [10], Finite Element Method (FEM) [37], Finite Volume Method (FEV) [11], ... Each of them presents its own advantages and disadvantages and are well established for dealing with nonlinearity, complex boundary conditions and complex geometries.

Overall, using these methods, the governing equations are adapted with a given boundary and initial conditions into a system of algebraic equations. These equations in turn can be solved following a large number of iterative or direct methods. There exist a broad range of iterative schemes with linear complexity for solving these equations such as Multigrid [62, 145] or preconditioned Krylov subspace methods [115]. The large size of the problems usually precludes conventional direct solvers based on Gaussian elimination since they have large memory requirement. However, direct solvers tend to be more stable and robust than iterative methods and in certain practical scenarios they are able to provide very fast and accurate solutions. In the first part of this thesis we have focused on studying the implementation on heterogeneous architectures of a Fast Direct Solver [154, 151], which has extensively used by the CFD community for solving the Poisson Equation.

These solvers, as many other software tools, have been developed to the point that many fluid engineering and scientific problems can now be computed routinely. However, although CFD is a well established discipline, there are still many challenges and open problems. Indeed, many groups are still developing new methods as well as alternative implementations to exploit the current and future developments in computer hardware.

In the second part of this thesis, we have focused on studying the Lattice Boltzmann Method (LBM), which is one of those relatively new emerging topics. Instead of solving the Navier–Stokes equations, the LBM is a clever discretization of the Boltzmann equation [135], that combines those characteristics developed to solve the Boltzmann equation over a finite number of microscopic speeds. The popularity of LBM has attracted interest from many researchers and their efforts have turned LBM into an alternative and promising numerical scheme for simulating complex fluid flows in different fields [86, 87, 16]. We have interest in studying its coupling with Immersed Boundary methods [102, 2, 56], being our focus on this thesis the mapping on heterogeneous architectures.

1.1.3. Investigated methods

Our target CFD solvers are in the incompressible flow domain. Despite the significant advances that have been achieved by state-of-the-art CFD methodologies, one aspect that still deserves further investigation is the acceleration of incompressible solvers, particularly when dealing with large problems with complex geometries. Using standard CFD methods, the computational cost of these kinds of solvers are dominated by the solution of the *Poisson equation for determining pressure* [58].

Our first contribution in this thesis has explored the acceleration on heterogeneous systems of fast Poisson solvers. The investigated method is easily accessible through a software package known as FISHPAK [48] (available at *netlib*), originally developed in the 1970s by Roland A. Sweet and Paul Swarztrauber from the National Center for Atmospheric Research (NCAR) [136] in Boulder, Colorado.

It combines the *Fast Fourier Transform* (FFT) with a *Generalized Block Cyclic Reduction* solver [136] and it is still considered as one of the most efficient solvers for the Poisson Pressure Equation developed so far. Our work has focused on the parallelization of FISHPACK' BLKTRI subroutine, which implements the generalized (extended) cyclic reduction solver. In Section 1.1.4 we review the related work and describe with more detail the solver implemented by the BLKTRI subroutine.

As we discuss in the first chapters of this thesis [154, 151], the speedups of our parallel implementation of the BLKTRI solver (the speedup over the baseline FISHPACK implementation) have been remarkable. However, our incompressible flow simulator still does not meet the performance goals envisioned. It is important to note that in a recent paper [122], Sudip K. Seal has claimed that other methods based on Recursive Doubling are better suited for heterogeneous platforms compared to our investigated variant of Cyclic Reduction. The main idea behind this claim is that Recursive Doubling is based on prefix scan primitives, which are efficiently supported on modern GPUs. However, even taking into account the potential improvement that we would achieve adopting this solver, we do not expect that such improvements would be high enough to compete with other emerging solvers in a massive parallel setting. This is why, in the second part of this work we opted to change our focus towards Lattice Boltzmann Methods (LBM), which are inherently more amenable for massively parallel architectures due to the high data independence that they presents [97].

As part of this work, we have developed a parallel LBM incompressible flow simulator from scratch, based on previous research codes developed by F. Julien et al. [41]. A performance evaluation of this solver is out of the scope of this thesis. Nevertheless, for the sake of completeness, Section 1.1.5 discusses in more detail

the main ideas behind LBM and reviews the state-of-the-art.

We have focused instead on studying the coupling of LBM with the Immersed-Boundary Method. The presence of an obstacle in the flow field, such as a solid body, can be managed by using *Boundary-Fitted* coordinate systems, which is a very complex and computationally expensive technique, see [11, 17]. However, other alternatives, such as the *Immersed-Boundary method* (IB) have emerged to mitigate these disadvantages [102]. IB computes the influence of the solids by distributing a set of markers (*Lagrangian* nodes) along their boundaries. Each node is formed by a subspace (support) of the *Cartesian* domain (fluid) over which two basic operations are carried out. These are *interpolation*, which computes the forces exerted over the *Lagrangian* nodes according to the values of the supports, and *spreading*, which propagates the forces over the *Cartesian* points. This new model (LBM-IB) has been analyzed in depth showing good numerical accuracy [41]. In the third part of this work, we have explored the mapping of IB-LBM methods on heterogeneous platforms. In Section 1.1.6 we review the state-of-the-art of this methodology.

1.1.4. Extended block cyclic reduction

Parallel Block Tridiagonal Solvers

There has been considerable work in developing efficient parallel solvers for linear systems of equations with scalar and block tridiagonal matrices. Here, we only briefly review some related work.

A block tridiagonal system of equations is represented by a matrix–vector equation of the form $Ax = b$ in which x and b are vectors of length N and A consists of

an $N \times N$ array of blocks where each block is an $M \times M$ array of numbers and the elements not belonging to its three central block diagonals are all zeros.

State-of-the-art software packages such as ScaLAPACK [27] or PETSc [129] implement very efficient solvers for dense and sparse matrices without any specific structure. However, it is possible to achieve higher performance and better scalability using specifically customized solvers that take advantage of the tridiagonality of the system matrix.

One of the algorithms that exploit this structure is a generalization of the Thomas algorithm (TA) [121]. TA is a simplified form of Gaussian elimination and obtains a direct solution with no fill-in. TA is the fastest sequential algorithm, but unfortunately, it is not parallelizable due to the inherent dependencies introduced by its forward and backward recurrences.

Tridiagonal direct solvers based on divide-and-conquer approaches were developed to overcome the aforementioned limitations of TA on parallel computers. Essentially, they are based on rearranging the computation to increase parallelism at the expense of introducing additional work. Cyclic Reduction (CR) is one of such divide-and-conquer algorithms; Other well-known alternatives are the Recursive Doubling (RD) algorithm, which was first introduced by Stone in [134], and the Partition Method developed by H. H. Wang [157]. Other studies have also investigated iterative methods, such as the Gauss-Seidel solver [9] or Krylov-based methods [65], that are out of the scope of this review.

CR was invented in the mid 1960s by G. H. Golub and R. W. Hockney for solving linear systems related to finite difference discretizations of the Poisson equation over a rectangle [69]. The basic idea of CR is that all the odd indexed unknowns of a tridiagonal linear system can be eliminated in terms of the even

indexed ones. The resulting system of equations is half the size of the previous one but with the same tridiagonal structure. Therefore, the same process can be cyclically repeated to the reduced system until a minimal number of equations are reached. The solution of this reduced system and back substitution complete the direct solver [18].

CR was extended and analyzed with more details some years later by Buneman, who provided a more stable version [23]. Since then it received much attention for its very nice computational features and had a great development [18]. These solvers are particularly amenable to efficient and scalable parallelization and many authors have explored the implementation of CR variants on all sort of parallel computing platforms [70]. Our work is based on its generalization to block tridiagonal forms, which, as mentioned above, were extensively studied by Roland A. Sweet and Paul Swarztraube [138, 140, 141, 137, 139].

For the sake of conciseness, we only review the most recent studies concerning the scalability and parallel performance of tridiagonal solvers on GPUs. Unfortunately, most of these studies have been limited to scalar versions of cyclic reduction and other well-know scalar parallel tridiagonal solvers [166, 53, 80]. Overall, they suggest that the most efficient approaches are based on hybrid solvers that combine TA with different CR variants or with the RD algorithm. However, these findings cannot be extrapolate to our target domain. Among the studies that have focused on block solvers, we should mention the works [68, 143, 123, 122], but none of them have investigated the acceleration of BLKTRI. Sudip K. Seal [122] has claimed that Recursive Doubling is better suited for heterogeneous platforms since it is based on prefix scan primitives, which are efficiently supported on modern GPUs.

The BLKTRI subrutine

The *BLKTRI* subroutine in the *FISHPACK* package [48] implements a classical direct method for the discrete solution of separable elliptic equations based on a block cyclic reduction algorithm. This method is commonly used when tackling the solution of a linear systems of equations arising from the second order centered finite difference discretization of 2D separable elliptic equations. From the standpoint of computational complexity (speed and storage), for a $m \times n$ net, its operation count is proportional to $mn \log_2 n$, and the storage requirements are minimal, since the solution is returned in the storage occupied by the right side of the equation (i.e., $m \times n$ locations are required). More in details, consider the 2D separable elliptic equation having $x(u,v)$ as an unknown field:

$$\frac{\partial}{\partial u}\left(a(u)\frac{\partial x}{\partial u}\right) + b(u)\frac{\partial x}{\partial u} + c(u)u + \frac{\partial}{\partial v}\left(d(v)\frac{\partial x}{\partial v}\right) + e(v)\frac{\partial x}{\partial v} + f(v)u = g(u,v) \tag{1.4}$$

If we discretize (1.4) with given Dirichlet or Neumann boundary conditions assigned on the edges of a square, using the usual five-point scheme with the discrete variables ordered in a lexicographic fashion, we obtain a linear system of $m \times n$ equations (having m nodes in the u direction and n in v): $\mathbf{A}\tilde{\mathbf{x}} = \tilde{\mathbf{g}}$, where $\mathbf{A}$ is a block tridiagonal matrix:

$$\mathbf{A} = \begin{bmatrix} B_1 & C_1 & & & & 0 \\ A_2 & B_2 & C_2 & & & \\ & . & . & . & & \\ & & . & . & . & \\ & & & A_{n-1} & B_{n-1} & C_{n-1} \\ 0 & & & & A_n & B_n \end{bmatrix}$$

and the vectors $\vec{x}$ and $\vec{g}$ are consistently split as a set of sub-vectors $\vec{x_i}$ and $\vec{g_i}$, $i = 1..m$, of length m each:

$$\mathbf{x} = [\tilde{\mathbf{x_1}},\ \tilde{\mathbf{x_2}}, \cdots\cdots \tilde{\mathbf{x_n}}]^{\mathbf{T}}$$

$$\mathbf{g} = [\tilde{\mathbf{y_1}},\ \tilde{\mathbf{y_2}}, \cdots\cdots \tilde{\mathbf{y_n}}]^{\mathbf{T}}$$

There is no restriction on m; however, cyclic reduction algorithms require $n = 2^k$, with large values of k for optimal performance. The blocks $\mathbf{A_i}$, $\mathbf{B_i}$ and $\mathbf{C_i}$ are $m \times m$ square matrices. In particular, the *BLKTRI* algorithm requires them to be of the form:

$$A_i = a_i I \tag{1.5}$$

$$B_i = B + b_i I \tag{1.6}$$

$$C_i = c_i I \tag{1.7}$$

where a_i, b_i and c_i are scalars. Having used a standard five point stencil for the discretization of (1.4), the matrix B is tridiagonal. The solution is obtained using an *extended cyclic reduction algorithm* which consists of the following phases:

1. Preprocessing phase: a set of intermediate results that only depend on the entries of **A**, not on the right hand side (*rhs*) of the equation, are obtained. Those results may be stored if a number of linear systems sharing the same coefficient matrix with different *rhs*, $\vec{g}$, needs to be solved.

2. Reduction phase: A sequence of linear systems are generated, starting from the original complete one, by decoupling odd and even equations. At each step, about half the unknown vector $\vec{x_i}$ are eliminated, with the result that each system has a block order of about half the former one. This process is continued until a system with the single unknown vector $\vec{x_2^k}$ is obtained.

3. Back-substitution phase: The solution vectors $\vec{x_i}$ are determined by first solving the final system generated in the above phase $\vec{x_2^k}$. Then the linear systems are solved in reverse order determining more $\vec{x_i}$ solution vectors, from the $\vec{x_i}$ previously computed.

During the reduction phase the following equations are tackled and solved [136]:

$$q_i^{(r)} = (B_i^r)^{-1} B_{i-2^{r-1}}^{r-1} B_{i+2^{r-1}}^{r-1} p_i^{(r)} \tag{1.8}$$

$$p_i^{(r+1)} = \alpha_i^r (B_{i-2^{r-1}}^{r-1})^{-1} q_{i-2^r}^{(r)} + \gamma_i^r (B_{i+2^{r-1}}^{r-1})^{-1} q_{i+2^r}^{(r)} - p_i^r \tag{1.9}$$

where **g** is split in two different terms, q and p. B stores the roots calculated in preprocessing phase. This procedure is required to stabilize the method [136]. α and γ have the following form:

$$\alpha_i^{(r)} = \prod_{j=i-2^{r+1}}^{i} aj \tag{1.10}$$

$$\gamma_i^{(r)} = \prod_{j=1}^{i+2^{r-1}} cj \tag{1.11}$$

Conversely, in the last phase the following equations are solved [136]: for $r = k, k-1, \ldots, 0$ and $i = 2^r, 3\times 2^r, 5\times 2^r, \ldots, 2^{k-r}\times 2^r$:

$$x_i = (B_i^r)^{-1} B_{i-2^{r-1}}^{r-1} B_{i+2^{r-1}}^{r-1} [p_i^{(r)} - \alpha_i^r (B_{i-2^{r-1}}^{r-1})^{-1} x_{i-2^r} - \gamma_i^r (B_{i+2^{r-1}}^{r-1})^{-1} x_{i+2^r}] \tag{1.12}$$

To obtain the aforementioned terms, the solution of a set of tridiagonal systems of equations must be addressed. The solution of these systems represents the most expensive stage of the algorithm. Also, other more basic mathematical operations such as vectors sums or scalar vector multiplications introduce a non negligible cost. The original sequential implementation of the algorithm in the FISHPACK package makes use of the TA algorithm [121] to tackle the solution of each tridiagonal problem.

TA consists of two stages, commonly denoted as forward elimination and backward substitution. Given a linear $Au = y$ system, where A is a tridiagonal matrix:

$$A = \begin{bmatrix} b_1 & c_1 & & & & 0 \\ a_2 & b_2 & c_2 & & & \\ & . & . & . & & \\ & & . & . & . & \\ & & & a_{n-1} & b_{n-1} & c_{n-1} \\ 0 & & & & a_n & b_n \end{bmatrix}$$

The forward stage eliminates the lower diagonal as follows:

$$c'_1 = \frac{c_1}{b_1}, \quad c'_i = \frac{c_i}{b_i - c'_{i-1} a_i} \quad \text{for } i = 2,3,\ldots,n-1$$
$$y'_1 = \frac{y_1}{b_1}, \quad y'_i = \frac{y_i - y'_{i-1} a_i}{b_i - c'_{i-1} a_i} \quad \text{for } i = 2,3,\ldots,n-1$$

and then the backward stage recursively solves each row:

$$u_n = y'_n, u_i = y'_i - c'_i u_{i+1} \text{ for } i = n-1, n-2, \ldots, 1$$

Overall, the complexity of TA is optimal: $8n$ operations in $2n-1$ steps, but as mentioned above, this algorithm is purely sequential.

CR [69] also consists of two phases (reduction and substitution). In each intermediate step of the reduction phase, all even-indexed (i) equations $a_i x_{i-1} + b_i x_i + c_i x_{i+1} = d_i$ are reduced. The values of a_i, b_i, c_i and d_i are updated in each step according to:

$$a'_i = -a_{i-1} k_1, b'_i = b_i - c_{i-1} k_1 - a_{i+1} k_2 \; c'_i = -c_{i+1} k_2, y'_i = y_i - y_{i-1} k_1 - y_{i+1} k_2$$

$$k_1 = \frac{a_i}{b_{i-1}}, k_2 = \frac{c_i}{b_{i+1}}$$

After $\log_2 n$ steps, the system is reduced to a single equation that is solved directly. All odd-indexed unknowns, x_i, are then solved in the substitution phase by introducing the already computed u_{i-1} and u_{i+1} values:

$$u_i = \frac{y'_i - a'_i x_{i-1} - c'_i x_{i+1}}{b'_i}$$

Overall, the CR algorithm needs $17n$ operations and $2\log_2 n - 1$ steps. Figure 1.2 graphically illustrates its access pattern.

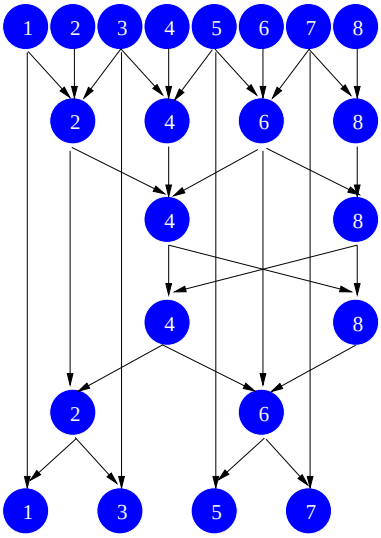


Figure 1.2: Access pattern of the CR algorithm.

Parallel Cyclic Reduction (PCR) [70] is a variant of CR, which only has a substitution phase. For convenience, we consider the case where $n = 2^s$, which involve $s = \log_2 n$ steps. Similarly to CR, a, b, c and y are updated as follows, for $j = 1, 2, \ldots, s$ and $k = 2^{j-1}$:

$$a'_i = \alpha_i a_i, b'_i = b_i + \alpha_i c_{i-k} + \beta_i a_{i+k}$$
$$c'_i = \beta_i c_{i+1}, y'_i = b_i + \alpha_i y_{i-k} + \beta_i y_{i+k}$$

$$\alpha_i = \frac{-a_i}{b_{i-1}}, \beta_i = \frac{-c_i}{b_i}$$

finally the solution is:

$$u_i = \frac{y'_i}{b_i}$$

Essentially, at each reduction stage, the current system is transformed into two smaller systems and after $\log_2 n$ steps the original system is reduced to n independent equations. Overall, the operation count of PCR is $12n\log_2 n$. Figure 1.3 sketches the corresponding access pattern.

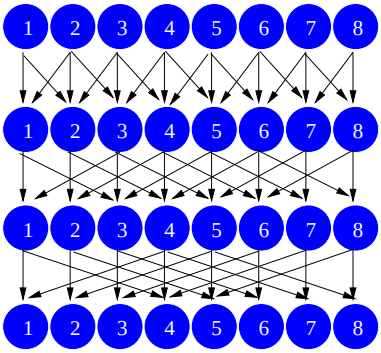


Figure 1.3: Access pattern of the PCR algorithm.

We should highlight that apart from their computational complexity these algorithms differ in their data access and synchronization patterns, which also have a strong influence on their actual performance. For instance, in the CR algorithm synchronizations are introduced at the end of each step and its corresponding memory access pattern may cause bank conflicts on modern GPUs. PCR needs less steps and its memory access pattern is more regular [166]. In fact, as mention above hybrid combinations that try to exploit the best of each algorithm have been explored, see [166, 116, 34, 80]. Figure 1.4 illustrates the access pattern of the CR-PCR combination proposed in [166]. CR-PCR reduces the system to a certain size using the forward reduction phase of CR and then solves the reduced (intermediate) system with the PCR algorithm. Finally, it substitutes the solved unknowns back into the original system using the backward substitution phase of CR.

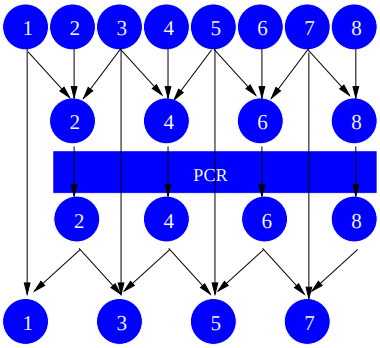


Figure 1.4: Communications pattern for the CR-PCR algorithm.

Using BLKTRI for solving 3D problems

Next, we extend the previous explanation to describe the strategy followed to solve a classical 3D Poisson equation:

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial z^2} = f(x,y,z)$$

defined on a Cartesian domain Ω with prescribed conditions on its boundary $\partial\Omega$.

Discretizing the domain using a uniform Cartesian mesh along each direction, for each (i,j,k) interior node we obtain:

$$\delta_x^2(i,j,k) + \delta_y^2(i,j,k) + \delta_z^2(i,j,k) = f_{i,j,k} \tag{1.13}$$

where

$$\begin{aligned}
\delta_x^2(i,j,k) &= \left(u_{i-1,j,k} - 2u_{i,j,k} + u_{i+1,j,k}\right)/\Delta x^2 \\
\delta_y^2(i,j,k) &= \left(u_{i,j-1,k} - 2u_{i,j,k} + u_{i,j+1,k}\right)/\Delta y^2 \\
\delta_z^2(i,j,k) &= \left(u_{i,j,k-1} - 2u_{i,j,k} + u_{i,j,k+1}\right)/\Delta z^2
\end{aligned}$$

are the finite difference centered approximations to the second derivatives along each direction. The boundary conditions which we will consider are either of Dirichlet or Neumann type on the surfaces normal to the y and z directions and periodic in the x-direction. The periodic condition applied in one of the directions allows to uncouple the 3D problem into a set of several independent 2D problems using a discrete Fourier transform. Hereafter we will briefly explain how the decoupling process takes place. Let N being the number of equispaced nodes in the x direction that cover the interval $(0, 2\pi)$. We expand the unknown function $u(x,y,z)$ and $f(x,y,z)$ in Fourier series as:

$$u_{n,j,k} = \frac{1}{N}\sum_{l=1}^{N} \hat{u}_{l,j,k} e^{-i\alpha(n-1)} \text{ with } \alpha = \frac{2\pi(l-1)}{N} \tag{1.14}$$

where $\hat{u}_{l,j,k}$ is the l^{th} Fourier coefficient of the expansion. Next, the expansion is applied to equation (3.1), obtaining the following relationship:

$$\frac{1}{N}\sum_{l=1}^{N} e^{-i\alpha(n-1)} \left\{ \frac{\hat{u}_{l,j,k}}{\Delta x^2}(e^{-i\alpha} - 2 + e^{i\alpha}) + \delta_y^2 \hat{u}_{l,j,k} + \delta_z^2 \hat{u}_{l,j,k} \right\} = \frac{1}{N}\sum_{l=1}^{N} \hat{F}_{l,j,k} e^{-i\alpha n} \tag{1.15}$$

Equation (3.3) is equivalent to the set of N equations ($l = 1 \cdots N$):

$$\frac{\hat{u}_{l,j,k}(2cos(\alpha) - 2)}{\Delta x^2} + \frac{\hat{u}_{l,j+1,k} - 2\hat{u}_{l,j,k} + \hat{u}_{l,j-1,k}}{\Delta y^2} + \frac{\hat{u}_{l,j,k+1} - 2\hat{u}_{l,j,k} + \hat{u}_{l,j,k-1}}{\Delta z^2} = \hat{F}_{l,j,k} \tag{1.16}$$

having used the identity $e^{i\alpha} + e^{-i\alpha} = 2cos(\alpha)$. In short notation (3.4) reads as follows:

$$\frac{\hat{u}_{l,j+1,k} + \hat{u}_{l,j-1,k}}{\Delta y^2} + \frac{\hat{u}_{l,j,k+1} + \hat{u}_{l,j,k-1}}{\Delta z^2} + \beta_l \hat{u}_{l,j,k} = \hat{F}_{l,j,k}, \;\; l = 1 \cdots N \tag{1.17}$$

with $\beta_l/2 = cos(\alpha) - 1/\Delta x^2 - 1/\Delta y^2 - 1\Delta z^2$. Thus, by considering the Fourier transform (direct FFT) of F one obtains a set of N, 2D independent problems having the Fourier coefficients $\hat{u}_{l,j,k}, l = 1..N$ as unknowns. Each independent problem is the solution of a linear system of equations where the coefficient matrix is block tridiagonal and can be solved using the *Extended Cyclic Reduction* approach. Once the solution is obtained, in Fourier space, a backward FFT can be used to recast the solution in physical space.

1.1.5. Lattice-Boltzmann method

Introduction and Related Works

Most of the current methods for simulating the transport equations (heat, mass, and momentum) are based on the use of macroscopic partial differential equations. On the other extreme, we can view the medium from a microscopic viewpoint where small particles (molecule, atom) collide with each other (molecular dynamic) [90]. In this scale the inter-particle forces must be identified, which requires one to know the location, velocity, and trajectory of every particle. However, there is no definition of viscosity, heat capacity, temperature, pressure, etc. These methods are extremely expensive computationally [90]. However, it is possible to use statistical mechanics as a translator between the molecular world and the microscopic world, avoiding the management of every individual particle, while obtaining the important macroscopic effects by combining the advantages of both macroscopic and microscopic approaches with manageable computer resources. This is the main idea of the Boltzmann equation and the mesoscopic scale [90].

Multiple studies have compared the efficiency of LBM with other methods (see [12, 81, 78, 109]). They show that LBM can achieve a similar numerical accuracy over a large number of applications as compared to the other methods. These kinds of solvers have important applications in bio-engineering applications [16].

Due to particular features of LBM, it has been adapted to numerous parallel architectures, such as multicore processors [104], manycore accelerators [110, 16, 29] and distributed-memory clusters [97, 73]. For instance, T. Pohl et al. [104] have focused on possible memory access patterns to maximize the temporal locality, optimizing the cache performance over multicore architectures. Also P. R. Rinaldi et al. [110] have modified the standard ordering of the LBM steps to reduce the number of memory accesses. Given the growing popularity of LBM, multiple tools [164, 101, 73] have recently arisen, consolidating this method into both academia and industry.

LBM formulation

In the last few decades the study of the relationship among Navier-Stokes equations and the Boltzmann equation has become an important research field [26, 131, 87]. The Boltzmann equation presents some relevant capabilities for modeling gas flows. However, numerical methods based on Navier-Stokes are more efficient, such that these methods are preferred for flow simulations. The first attempts towards a simplified approach of the Boltzmann equation, such as the lattice-gas automata [50, 38], introduced a new approach for simulating fluid flow in an efficient way. Today, these new solvers [66, 67] have become a real alternative to classical fluid-flow approaches based on Navier-Stokes.

Lattice-Boltzmann methods (LBM) combine those characteristics developed to solve the Boltzmann equation over a finite number of microscopic speeds. LBM presents some lattice-symmetry properties which allows the conservation of the macroscopic moments [63]. The standard Lattice-Boltzmann method [107] is an explicit-time-step solver for incompressible flows. It divides each temporal iteration into two steps, one for propagation-advection and one for collision step which represents inter-particle interactions, achieving a first order in time and second order in space scheme.

LBM describes the fluid behavior at mesoscopic level. At this level, the fluid is modeled by a distribution function of the microscopic particle, f. Similarly to the Boltzmann equation, LBM solves the particle speed distribution by discretizing the speed space over a discrete finite number of possible speeds. The distribution function evolves according to the following equation:

$$\frac{\partial f}{\partial t} + e\nabla f = \Omega \tag{1.18}$$

where f is the particle distribution function, e is the discrete space of speeds and Ω is the collision operator. By discretizing the distribution function f in space, in time, and in speed ($\mathbf{e} = \mathbf{e}_i$) we obtain $f_i(\mathbf{x},t)$, which describes the probability of finding a particle located at $\mathbf{x}$ at time t with speed $\mathbf{e}_i$.

The term $e\nabla f$ can be discretized as:

$$e\nabla f = e_i \nabla f_i = \frac{f_i(\mathbf{x} + e_i\Delta t, t + \Delta t) - f_i(\mathbf{x}, t + \Delta t)}{\Delta t} \tag{1.19}$$

In this way the particles can move only along the links of a regular Lattice (Figure 1.5) defined by the discrete speeds ($e_0 = c(0,0); e_i = c(\pm 1, 0),\ c(0, \pm 1),$

$i = 1, \cdots, 4$; $e_i = c(\pm 1, \pm 1)$, $i = 5, \cdots, 8$ with $c = \Delta x / \Delta t$) so that the synchronous particle displacements $\Delta \mathbf{x}_i = \mathbf{e}_i \Delta t$ never takes the fluid particles away from the Lattice. For clarify, the standard two-dimensional 9-speed lattice *D2Q9* is considered [63], but all the techniques which will be presented here can be extended, in a straightforward manner, to the standard three dimensional lattice *D3Q19*.

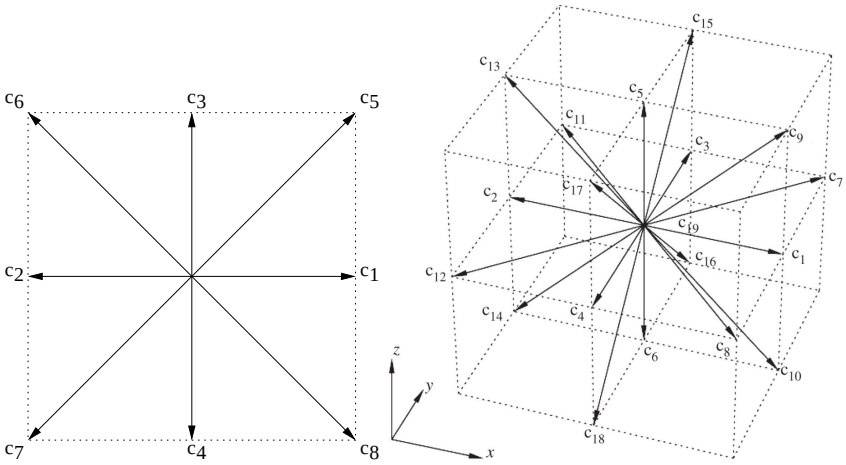


Figure 1.5: The standard two-dimensional 9-speed lattice (*D2Q9*) used (left) and the standard three-dimensional 19-speed lattice (*D3Q19*) [163].

The operator Ω describes the changes suffered by the collision of the microscopic particles, which affect the distribution function f. To calculate the collision operator we consider the *BGK* (Bhatnagar-Gross-Krook) formulation [100] which relies upon a unique relaxation time, τ, toward the equilibrium distribution f_i^{eq}:

$$\Omega = -\frac{1}{\tau}\left(f_i(\mathbf{x},t) - f_i^{eq}(\mathbf{x},t)\right) \tag{1.20}$$

The equilibrium function $f^{eq}(\mathbf{x},t)$ can be obtained by Taylor series expansion of the Maxwell-Boltzmann equilibrium distribution [107]:

$$f_i^{eq} = \rho\,\omega_i \left[1 + \frac{\mathbf{e_i}\cdot\mathbf{u}}{c_s^2} + \frac{(\mathbf{e_i}\cdot\mathbf{u})^2}{2c_s^4} - \frac{\mathbf{u}^2}{2c_s^2}\right] \tag{1.21}$$

where c_s is the speed of sound ($c_s = 1/\sqrt{3}$) and the weight coefficients ω_i are $\omega_0 = 4/9,\ \omega_i = 1/9,\ i = 1\cdots4$ and $\omega_5 = 1/36,\ i = 5\cdots8$ based on the current normalization. Through the use of the collision operator and substituting the term $\frac{\partial f_i}{\partial t}$ with a first order temporal discretization, the discrete Boltzmann equation can be written as:

$$\frac{f_i(\mathbf{x},t+\Delta t) - f_i(\mathbf{x},t)}{\Delta t} + \frac{f_i(\mathbf{x}+e_i\Delta t, t+\Delta t) - f_i(\mathbf{x},t+\Delta t)}{\Delta t} = -\frac{1}{\tau}\left(f_i(\mathbf{x},t) - f_i^{eq}(\mathbf{x},t)\right) \tag{1.22}$$

which can be compactly written as:

$$f_i(\mathbf{x}+\mathbf{e}_i\Delta t, t+\Delta t) - f_i(\mathbf{x},t) = -\frac{\Delta t}{\tau}\left(f(\mathbf{x},t) - f_i^{eq}(\mathbf{x},t)\right) \tag{1.23}$$

The macroscopic velocity $\mathbf{u}$ in equation 1.21 must satisfy a Mach number requirement $\mid \mathbf{u} \mid /c_s \approx M << 1$. This stands as the equivalent of the CFL number[1] for classical Navier Stokes solvers.

As mentioned above, the equation 1.23 is typically advanced in time in two stages, the collision and the streaming stages.

Given $f_i(\mathbf{x},t)$ compute:

$\rho = \sum f_i(\mathbf{x},t)$ and

$\rho\mathbf{u} = \sum \mathbf{e}_i f_i(\mathbf{x},t)$

[1]Courant–Friedrichs–Lewy (CFL) number arises in those schemes based on explicit time computer simulations. As a consequence, this number must be less than a certain time to achieve coherent results.

Collision stage:

$$f_i^* (\mathbf{x}, t + \Delta t) = f_i (\mathbf{x}, t) - \frac{\Delta t}{\tau} \left(f (\mathbf{x}, t) - f_i^{eq} (\mathbf{x}, t) \right)$$

Streaming stage:

$$f_i (\mathbf{x} + \mathbf{e}_i \Delta t, t + \Delta t) = f_i^* (\mathbf{x}, t + \Delta t)$$

LBM solvers

Algorithm 1 shows a possible approach of the steps which compose LBM. For computing the streaming step, in parallel, we considere two different distribution functions. LBM exhibits a higher degree of parallelism than traditional solvers, which can be very appropriate for parallel computer architectures, by exploiting a fine granularity since every lattice point is totally independent.

As introduced above, LBM can be implemented following a fine granularity scheme (one thread per lattice node). However, depending on the ordering of the collision and streaming stages two different strategies arise: The classical approach is known as the *push* method and performs the *collide* step before the *streaming* step, on the contrary, the *pull* approach performs the steps in the opposite order. These differences have important consequences in terms of performance and efficiency.

The computational scheduling of LBM based on *push* approach (*collide-stream* strategy) has been used in numerous works (see [167, 119, 16]). In general, the *push* method divides the LBM steps into two steps. The first one computes the *collide* and *stream* phases and the second one computes the macroscopic variables (velocities and density). This degrades the adaptation of the algorithm to parallel architectures

Algorithm 1 LBM implementation.

```
1: Macroscopic Level
2: for ind = 1 → Nx·Ny do
3:     for i = 1 → 9 do
4:         ρ[ind] += f1[i][ind]
5:         ux[ind] += cx[i]·f1[i][ind]
6:         uy[ind] += cy[i]·f1[i][ind]
7:     end for
8: end for
9: ux[ind] = ux[ind]/ρ[ind]
10: uy[ind] = uy[ind]/ρ[ind]
11: Stream
12: for ind = 1 → Nx·Ny do
13:     for i = 1 → 9 do
14:         x_stream = x + cx[i]
15:         y_stream = y + cy[i]
16:         ind_stream = y_stream·Nx + x_stream
17:         f2[i][ind_stream] = f1[i][ind]
18:     end for
19: end for
20: Collision
21: for ind = 1 → Nx·Ny do
22:     for i = 1 → 9 do
23:         cu = cx[i]·ux[ind] + cy[i]·uy[ind]
24:         f_eq = ω[i]·ρ[ind]·(1 + 3·cu + cu^2 − 1.5·(ux[ind])^2 + uy[ind])^2))
25:         f1[i][ind] = f2[i][ind]·(1 − 1/τ) + f_eq·1/τ
26:     end for
27: end for
```

and imposes greater pressure on memory (a sketch of this LBM scheduler is given in Algorithm 1).

The *pull* computational scheduling scheme (introduced by [158]) has been recently consider in [110]. This is an efficient approach based on a single-loop strategy; Each lattice node can be independently computed by performing one complete time step of LBM (a schematic sketch of this LBM implementation is given in Algorithm 2). Basically, this strategy does not need any synchronization among the LBM steps, and thus is very profitable for parallel architectures. Furthermore, it

Algorithm 2 LBM *pull*.

1: **for** $ind = 1 \rightarrow Nx \cdot Ny$ **do**
2: **for** $i = 1 \rightarrow 9$ **do**
3: $x_{stream} = x - c_x[i]$
4: $y_{stream} = y - c_y[i]$
5: $ind_{stream} = y_{stream} \cdot Nx + x_{stream}$
6: $f[i] = f_1[i][ind_{stream}]$
7: **end for**
8: **for** $i = 1 \rightarrow 9$ **do**
9: $\rho += f[i]$
10: $u_x += c_x[i] \cdot f[i]$
11: $u_y += c_y[i] \cdot f[i]$
12: **end for**
13: $u_x = u_x/\rho$
14: $u_y = u_y/\rho$
15: **for** $i = 1 \rightarrow 9$ **do**
16: $cu = c_x[i] \cdot u_x + c_y[i] \cdot u_y$
17: $f_{eq} = \omega[i] \cdot \rho \cdot (1 + 3 \cdot cu + cu^2 - 1.5 \cdot (u_x)^2 + u_y)^2))$
18: $f_2[i][ind] = f[i] \cdot (1 - \frac{1}{\tau}) + f_{eq} \cdot \frac{1}{\tau}$
19: **end for**
20: **end for**

eases pressure on memory with respect to the *push* approach, as the macroscopic level can be completely computed on top regions of memory hierarchy.

Given the characteristics of the push and pull strategies mentioned above, we have opted to use the pull scheduling in our solvers.

1.1.6. Solid-fluid interaction based on the Lattice-Boltzmann method and immersed boundary method coupling

Introduction and related works

Solid-fluid interaction is a research topic currently enjoying growing interest in many scientific communities; It is intrinsically interdisciplinary (structural mechanic, fluid mechanic, applied mathematics, etc) and covers a broad range of

applications (aeronautics, civil engineering, biological flows, etc). The number of works on this topic reflects the growing importance of the study of the dynamics solid/s [165, 24, 112, 84, 147, 95, 99].

Similar to the original work on the mathematical formulation of the Immersed Boundary (IB) algorithm presented by Peskin [102], we have considered an IB approach based on the work of M. Uhlmann [148] to enforce the presence of a solid on the fluid field. The main goal of IB consists of handling complex geometries in Cartesian grids. It requires a low computational effort without sacrificing too much solution accuracy. Other conventional approaches for solid-fluid interaction such as Boundary-Fitted coordinate systems [11, 17] and the Cut-Cell methods [4, 77] present several inconveniences for dealing with complex geometries, moving and flexible bodies. These approaches are complex and computationally expensive. In contrast, those approaches based on IB exhibit more advantages with respect to memory and run time savings.

IB is well established and has been used in numerous complex configurations, such as complex geometries, moving and deformable solids, etc, with satisfactory results [168, 169, 148, 3]. We have focused on the optimization of this method due to the particular characteristics which IB presents, and the large range of applications where it can be used. Furthermore, this is presented as an efficient, accurate and computationally cheap choice for this type of configuration. Several versions of the IB method have been developed in response to the needs of their application: in addition to the original version, there exist other approach, such as the vortex-method [88], volume-conserved [103], mesh adaptavity [2], (formally) second-order [56], multigrid [168], amongst others.

Solid-fluid interaction based on IB methods has only recently gained wider

interest in high-performance computing. Other authors address topics which are somehow related to our contribution; A recent work is the one by H. Zhou et al. [167] which shows a Lattice-Boltzmann based implementation with curved boundary, where a flow around a circular cylinder is tested as a typical case. Curved boundaries are taken into account via a non equilibrium extrapolation scheme. H. Ji et al. [76] present a GPU-based implementation for solid-fluid interaction based on the coupling of adaptive mesh refinement methods and IB. S. K. Layton et al. [114] have studied the implementation on GPUs of the *IB projection* method introduced in [142] for the solution of two-dimensional incompressible viscous flows with immersed boundaries. Their numerical framework is based on a Navier-Stokes solver and they used the open-source `Cusp` library. Our method is a different approach based on the *different forcing* approach [148] which is able to deal with complex, moving, or deformable boundaries, and has been used on Lattice-Boltzmann [41] and Navier-Stokes [3] solvers.

Immersed boundary method

The basic idea behind this method consists of splitting the time advancement of the fluid momentum equation into two stages: the first without any body forces (no solids) and the second one, adding to the right hand side, a body force which restores the zero velocity boundary condition on the solid surfaces. The core of IB consists of computing these body forces. The fluid is discretized on a regular Cartesian mesh while the shape of the solids are discretized in a *Lagrangian* fashion, by a set of points which obviously do not necessarily coincide with mesh points. This is sufficient information to impose the body forces on the solid surface.

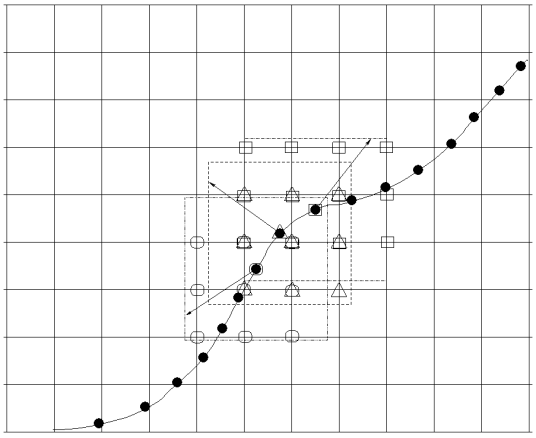


Figure 1.6: An immersed curve discretized with *Lagrangian* points denoted in the graph as •. Three consecutive points are considered with the respective supports.

It is necessary to compute these forces on a support, a set of Cartesian points around each *Lagrangian* (solid surface) point. The support of continuous *Lagrangian* points of the same solid share several Cartesian points (Figure 1.6). The key aspects of the algorithm are the interpolation I and the S operators (termed as spread from now on). Here, we perform both operations (interpolation and spread) through a convolution with a compact support mollifier meant to mimic the action of a Dirac's delta. Combining the two operators we can write in a compact form:

$$\mathbf{f}_{(ib)}(\mathbf{x},t) = \frac{1}{\Delta t}\int_{\Gamma}\left(\mathbf{U}^{\mathbf{d}}(\mathbf{s},t+\Delta t) - \int_{\Omega}\hat{\mathbf{u}}(\mathbf{y})\tilde{\delta}(\mathbf{y}-\mathbf{s})d\mathbf{y}\right)\tilde{\delta}(\mathbf{x}-\mathbf{s})d\mathbf{s} \qquad (1.24)$$

where $\tilde{\delta}$ is the mollifier, Γ is the set of *Lagrangian* points (immersed boundary), Ω is the computational domain, and $\mathbf{U}^{\mathbf{d}}$ is the desired value on the boundary at the next time step. The discrete equivalent of Equation 1.24 is simply obtained by any standard composite quadrature rule applied on the union of the supports associated to each *Lagrangian* point. As an example, the quadrature needed to obtain the force

distribution (spreading) on the Cartesian nodes is given by:

$$f^{ib}(x_i, y_j) = \sum_{n=1}^{N_e} F^{ib}(\mathbf{X_n})\tilde{\delta}(x_i - X_n, y_j - Y_n)\varepsilon_n \tag{1.25}$$

(x_i, y_j) are the Cartesian nodes falling within the union of all the supports, N_e is the number of *Lagrangian* points and ε_n is a value to be determined to enforce consistency between interpolation and the spreading (Equation 1.25). More details about the method and in particular about the determination of the ε_n values can be found in [3]. In what follows we will give more details on the construction of the support cages surrounding each *Lagrangian* point, since it plays a key role in the parallel implementation of the IB method. As already mentioned, the solid surfaces are discretized into a number of *Lagrangian* points $\mathbf{X}_I$, $I = 1..N_e$. Around each point $\mathbf{X}_I$ we define a rectangular cage Ω_I with the following properties: (i) it must contain at least three nodes of the underlying Eulerian mesh for each direction, (ii) the number of nodes contained in the cage must be minimized. The modified kernel, obtained as a Cartesian product of the one dimensional function [2]:

$$\tilde{\delta}(r) = \begin{cases} \frac{1}{6}\left(5 - 3|r| - \sqrt{-3(1-|r|)^2 + 1}\right) & 0.5 \leq |r| \leq 1.5 \\ \frac{1}{3}\left(1 + \sqrt{-3r^2 + 1}\right) & 0.5|r| \leq 0.5 \\ 0 & \text{otherwise} \end{cases} \tag{1.26}$$

which will be identically zero outside the square Ω_I. We take the edges of the square to measure slightly more than three spacings, Δ. With such a choice, at least three nodes of the mesh in each direction fall within the cage. The interpolation stage is performed locally on every point which composes the set of supports;

the values of velocity at the nodes within the support cage centered around each *Lagrangian* point delivering approximate values (i.e., second order) of velocity at the point location. The force spreading step requires information from all the points. The collected values are then distributed on the union of the supports meaning that each support may receive information from supports centered about neighboring points, as in Equation 1.25. Finally, the complete set of steps for the IB method are briefly described. Let the superscript * refer to the predicted variables without solid influences, $V\_(C\_)Lg^i_{x(y)}$ and $C\_Sp^j_{x(y)}$ the horizontal (x) and the vertical (y) velocities (V), the coordinates (C) for the i^{th} *Lagrangian* (Lg) point and the j^{th} support (Sp) point:

1. **Compute the U^*_x and U^*_y fields without solid forces**. These fields can be computed according to several algorithms such as the solvers based on Navier-Stokes (FDM, FEM, FEV, ...) or Lattice-Boltzmann.

2. **Velocity Interpolation (fluid $\rightarrow$ solid)**

$$\begin{aligned} V\_Lg^i_x &+= I(U^*_x[C\_Sp^j_x]) \\ V\_Lg^i_y &+= I(U^*_y[C\_Sp^j_y]) \\ &\forall i \in N_e, \forall j \in \Omega_i \end{aligned} \tag{1.27}$$

3. **Compute the forces on the solid surface (*Lagrangian* points)**.

$$\begin{aligned} F^{ib}_x(\mathbf{X_i}) &= \mathbf{U^d_x} - V\_Lg^i_x \\ F^{ib}_y(\mathbf{X_i}) &= \mathbf{U^d_y} - V\_Lg^i_y \\ &\forall i \in N_e \end{aligned} \tag{1.28}$$

4. **Spread the forces (solid $\rightarrow$ fluid)**.

$$\begin{aligned} f_x^{ib}(C\_Sp_x^j, C\_Sp_y^j) += S(F_x^{ib}(\mathbf{X_i})) \\ f_y^{ib}(C\_Sp_x^j, C\_Sp_y^j) += S(F_y^{ib}(\mathbf{X_i})) \\ \forall i \in N_e, \forall j \in \Omega_i \end{aligned} \tag{1.29}$$

5. **Adding the body forces to the U^* fields**. Depending of the method used to compute U^*, the f_{ib} is added.

$$\begin{aligned} U_x(C\_Sp_x^j, C\_Sp_y^j) += f_x^{ib}(C\_Sp_x^j, C\_Sp_y^j) \\ U_y(C\_Sp_x^j, C\_Sp_y^j) += f_y^{ib}(C\_Sp_x^j, C\_Sp_y^j) \\ \forall i \in N_e, \forall j \in \Omega_i \end{aligned} \tag{1.30}$$

LBM-IB coupling

In the following, we described the LBM-IB coupling. The LBM combined with an IB method is highly attractive when dealing with solids for two main reasons: the shape of the boundary, tracked by a set of *Lagrangian* nodes, is sufficient information to impose the boundary values; and the force of the fluid on the immersed boundary is readily available, and thus easily incorporated in the set of equations which govern the dynamics of the immersed object. In addition, it is also particularly well suited for massively parallelized simulations, as the time advancement is explicit and the computational stencil is formed by few local neighbors of each computational node (support). In what follows, we briefly recall the basic formulation of the coupled methods.

First of all, we analyze the incorporation of the IB forces to lattice level. This

is carried out through the computation of a set of forces (lattice forces), which link the IB forces to be included in our LBM solver. These lattice forces represent the contribution of external volume forces at lattice level, which, in our case, include the effect of the immersed boundary. Given any external volume force $\mathbf{f}^{(ib)}(\mathbf{x},t)$, the contribution on the lattice are computed according to the formulation proposed by [59] as:

$$F_i = \left(1 - \frac{1}{2\tau}\right) \omega_i \left[\frac{\mathbf{e_i} - \mathbf{u}}{c_s^2} + \frac{\mathbf{e_i} \cdot \mathbf{u}}{c_s^4}\mathbf{e_i}\right] \cdot \mathbf{f_{ib}} \tag{1.31}$$

Next, we briefly explain the numerical approach, in which the IB method is used, both to enforce boundary values, and to recover the fluid force exerted on immersed objects within the framework of the LBM algorithm. The general setup of the present LBM-IB solver can be recast in the following algorithmic sketch.

Given $f_i(\mathbf{x},t)$ compute:

Collision stage:

$\hat{f}_i^* (\mathbf{x}, t+\Delta t) = f_i(\mathbf{x},t) - \frac{\Delta t}{\tau}\left(f(\mathbf{x},t) - f_i^{(eq)}(\mathbf{x},t)\right)$

Streaming stage:

$\hat{f}_i(\mathbf{x}+\mathbf{e}_i\Delta t, t+\Delta t) = \hat{f}_i^*(\mathbf{x}, t+\Delta t)$

Compute :

$\hat{\rho} = \sum \hat{f}_i(\mathbf{x}+\mathbf{e}_i\Delta t, t+\Delta t)$ and

$\hat{\rho}\hat{\mathbf{u}} = \sum \mathbf{e}_i \hat{f}_i(\mathbf{x}+\mathbf{e}_i\Delta t, t+\Delta t)$

Interpolate on Lagrangian markers (volume force):

$\hat{\mathbf{U}}(\mathbf{X}_k, t+\Delta t) = I(\hat{\mathbf{u}})$ and

$\mathbf{f_{ib}}(\mathbf{x},t) = \frac{1}{\Delta t} S\left(\mathbf{U}_d(\mathbf{X}_k, t+\Delta t) - \hat{\mathbf{U}}(\mathbf{X}_k, t+\Delta t)\right)$

Compute lattice forces (see equation 1.31):

$$F_i = \left(1 - \frac{1}{2\tau}\right) \omega_i \left[\frac{\mathbf{e_i} - \mathbf{u}}{c_s^2} + \frac{\mathbf{e_i} \cdot \mathbf{u}}{c_s^4} \mathbf{e_i}\right] \cdot \mathbf{f_{ib}}$$

Repeat collision with lattice forces and Streaming:

$$f_i^* (\mathbf{x}, t + \Delta t) = f_i (\mathbf{x}, t) - \frac{\Delta t}{\tau} \left(f (\mathbf{x}, t) - f_i^{(eq)} (\mathbf{x}, t) \right) + \Delta t F_i \text{ and}$$

$$f_i (\mathbf{x} + \mathbf{e}_i \Delta t, t + \Delta t) = f_i^* (\mathbf{x}, t + \Delta t)$$

As outlined above, the basic idea consists in performing each time-step twice: The first one, performed without solid input, allows one to predict the velocity values at the immersed boundary markers and the force distribution that restores the desired velocity boundary values at their locations; The second one applies to the regularized set of singular forces and repeats the procedure advancing (using Eq. 1.31) to determine the final values of the distribution function at the next time-step.

1.2. Heterogeneous computing

In this section we summarize the technological trends that have led to the use of heterogeneous computing and describe CUDA, a new programming languages developed for nVidia for its GPUs that inspired the new OpenCL standard. We also review the evolution of parallel computing since many of the design principles behind current architectures were explored in the past.

1.2.1. The switch to multicore and multithreaded architectures

In 1965, one of the Intel's co-founder, Gordon Moore, realized that, with the advances achieved in the integration technology, it was economically possible to double the number of transistors per chip every 18 months [91]. This prediction, known later as "Moore's Law", has been confirmed until today and it is expected to hold true for the next few years [49].

Gordon Moore never said anything about processing performance when he made his prediction. His observation only links integration levels to production cost [127]. However, a few years later than Moore, Robert H. Dennard and his colleagues from IBM, articulated a set of rules (Dennard scaling) that link transistor size with perfomance and power[35]. The key observation they made was that smaller transistors can switch quickly at lower supply voltages, resulting in more power efficient circuits and keeping the power density constant [127]. For about four decades, Moore's law coupled with Dennard scaling have enabled that every technology generation have more transistors that are, not only smaller, but also much faster and more energy efficient.

This has allowed computer architects to increase the performance of processors, even without increasing their area and power. Indeed, during the 80's and early 90's, actual processor performance increased faster than Moore's law and Dennard scaling predicted [64]. The surplus of transistors was used by computer architects to integrate complex techniques to hide memory access latency and extract instruction level parallelism (ILP). Out-of-order execution, branch prediction and speculative execution, register renaming, non-blocking caches or memory dependence prediction were some relevant examples. Notoriously, all of them were

able to improve performance while maintaining the conventional Von Neumann computational mode. In other words, these techniques were virtually invisible to software. In this way, most applications improved their performance as technology scaled without needing to rewrite them [64].

Unfortunately, Dennard scaling appears to have ended since technology scaling broke the 100 nm barrier (around 2004-07). Essentially, supply voltages cannot drop forever because sub-threshold leakage currents are exponential to threshold voltage reductions. Overall, when threshold voltages dropped low enough, static power consumption become a major issue [127]. Although the number of transistors per chip continues to double roughly every two years [49], it has become increasingly difficult to continue to improve the performance:

- ***Frequency scaling*** **has stalled due to cooling and power concerns**. Some one-time reductions of static power consumption are still possible. For instance, current technologies employ multi-gate transistors also known as FinFETs (Intel switched to 3D or tri-gate transistors in their 22 nm technology). However, further reductions will be limited in subsequent scalings [127].

- **The cost of *Memory access* has continued to increase** and is now quite high relative to the cost of computation. Caches mitigate the memory wall but are of limited use for data intensive applications unless the entire dataset can fit in the cache. In fact, caches already occupy over half the silicon area of some processors (see figure 1.7) and consume much of the power [51]. As an alternative, some modern architectures rely on hardware multithreading such as *blocking* or *interleaved* multithreading to hide memory accesses, but they usually worsen single-threaded performance [51].

- ***ILP techniques* do not scale either**. Unfortunately, typical instruction streams have only a limited amount of usable parallelism among instructions [98]. Techniques such as *Simultaneous multithreading* [146], i.e. allowing different threads to execute simultaneously issue instructions on independent functional units, are able to improve the efficiency of superscalar processors without having to find ILP within a single thread. However, in most cases SMT only improves throughput at may decrease single thread performance when there is contention for shared resources. In fact, software developers usually have to test whether SMT improve performance on their application. Some researchers [111, 108] have explored techniques for using additional threads (usually known as helper or assistant threads) to speed up single-thread workloads. These threads can act as software-guided prefetchers seeding a shared resource like a cache or to provide early branch resolution. Unfortunately, the benefits of these proposals are limited and they have not materialize yet into commercial processors or compiler tools.

The breakdown of Dennard scaling prompted the switch to multicore and multithreaded architectures that we have experience over the last decade. Broadly speaking, the semiconductor industry has abandoned complex cores in favor of integrating more cores on the same chip [52]. Further, hardware multithreading has become essential to mask long-latency operations such as main memory accesses [92]. The assumption of this new paradigm is that as the number of processors/threads on a chip doubles, the performance of scalable parallel program will also continue to improve. However, there is now a major problem at the software side. In contrast to previous generations, programmers are in charge of exposing the parallelism

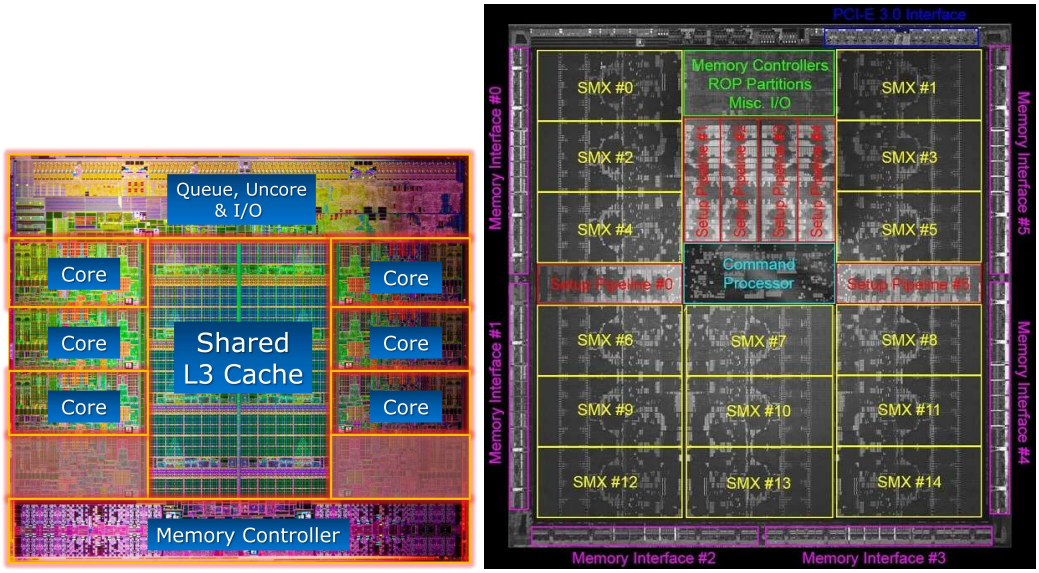


Figure 1.7: Die micro-photograph of an Intel Core i7 3960X-6 multicore processor (left) and a nVidia Tesla GPU K20 (right). A large fraction of the silicon area in multicore processors is occupied by caches, while gpus rely on hardware multithreading to hide memory accesses.

in their applications and need it to improve performance. This is not a simple task. Despite more than 40 years' experience with parallel computers, we know that parallel programs are usually difficult to design, implement and debug, and their performance do not always scale well with the number of cores/threads. As Gene Amdahl observed as early as in 1967 [8], we are unable to efficiently extract sufficient parallelism from many applications. In other words, the gains offered by switching to more cores are in practice much lower than the gains that would be achieved had Dennard scaling continued, but unfortunately technology do not offer us any alternative.

The first general-purpose processor that included multiple processing cores on the same die was released by IBM in 2001 (the IBM POWER4 processor) [98]. Since then, multi-core processors have become the norm. In fact, the major way

to improve the performance of high-end processors has been to add support for more threads, either increasing the number of cores per chip or through hardware multithreading [150]. This new trend is well exemplified by modern GPUs [85], with rely on fine-grain multithreading coupled with simple processing cores and SIMD execution to maximize performance when parallelism is abundant [51].

1.2.2. The evolution of parallel computing

Multi-core chips were a new paradigm when introduced a decade ago, but parallel computing is not new. In fact, parallelism has always been a means to satisfy our never-ending hunger for ever-faster and ever-cheaper computation [36] and many of the design principles behind current architectures were used in the past.

The very first multiprocessor architecture was the Burroughs B5000, which was designed in the early 60's [36]. This machine, along with its successors, used shared memory multiprocessors in which a crossbar switch connected groups of four processors and memory boxes [36].

Computers containing multiple processors sharing a common memory dominate the server and mainframe markets in the mid 80's. The early systems were introduced by famous small companies such as Encore and Sequent [31]. The early 1990s brought a dramatic advance in the shared memory bus technology, including faster electrical signalling, wider data paths, pipelined protocols, and multiple paths [31]. Distributed memory systems and massive parallel processors (MPPs) used complex networks that allow higher scalability [31]. An evolution of these types of interconnects can be found today in multicore chips [117].

The ILLIAC IV [14], designed during the late 60's and delivered to NASA Ames Research Center in 1971, was one of the first attempts to build a SIMD array processor. The project involved the University of Illinois and Burroughs and was based on the earlier Solomon work [31] (and in spite of the famous Amdahl's arguments to the contrary [8]). Other supercomputers of the era, such as the classic Cray-1 [113], used instead a single vector processor with multiple pipelined functional units.

Technology scaling overtook the specialized SIMD and vector processor in favor of MPPs such as the Intel Paragon [40] or the Cray T3E [120] and cluster of workstations (COWs) in the 90's. The appearance of Beowulf clusters, originally developed by Thomas Sterling and Donald Becker at NASA [132], provided considerable computational resources using commodity hardware components such as PCs and Ethernet switches. The reduced economic cost of such systems paved the way for the popularization of parallel computing since many university labs and research centers can afford them.

The peripheral processor of the Control Data Corp (CDC 6600) developed in the 1960s and the Heterogeneous Element Processor system developed in the late 1970s [128] are notable examples of the early use of multithreaded architectures. In such architectures, a single processor has the ability to follow multiple streams of execution without the aid of software context switches that require many thousands of cycles [92]. A multithreaded architecture can access the state of multiple thread, which allows it to quickly switch between threads. Several models of multithreading have been explored and implemented since then [149].

Prototypes such as Imagine [1], Merrimac [32], and SPI Storm [79] exploit *blocking multithreading*. This is a coarser-grain strategy in which a thread continue running until encountering a long-latency operation, at which point a different

thread is selected for execution. These machines explicitly partition programs into blocks of high-latency memory access (load/store) operation and tasks in which memory accesses are restricted to on-chip (local) memory. When a block finishes processing its on-chip data, a different block, which required high-latency memory accesses, have been loaded onto the chip is executed. Overlapping the blocks data transfer for one or more tasks while another is executing hides memory-access latency.

The Tera [7, 6] prototype and the Sun Niagara [82] processors used *fine-grain multithreading* to hide long latency operations at the expense of sacrificing single-threaded performance. These processors are able to switch between threads at finer granularity than blocking multithreading (even at each clock cycle), achieving impressive performance for workloads in which parallelism is abundant. As mentioned above, this multithreading technique have evolved into the architecture used in modern GPUs today [51].

1.2.3. The rise of heterogeneous computing

Despite much progress, multicore designs are also encountering scaling problems, notably the "Dark Silicon" phenomenon [39]. Power and cooling concerns suggest the number of dynamically active transistors on a single die may be greatly constrained in the near future. In other words, even if the number of transistors per chip continues to follow Moore's law, we will not be able to use all of them simultaneously. This problem may lead to scenarios in which only a small percentage of the chip's transistors can be "on" at a time [127].

Heterogeneous architectures may be an answer to this challenge. In these architectures, some general-purpose cores are augmented by other cores that implement different microarchitectures or even specialized accelerators that are more efficient for a particular computational purpose [127]. Again, the main problem is at the software side. Programmers need to address the difficult optimization challenge of choosing the right processor for different parts of their applications in order to achieve the best performance or performance-per-watt on those complex heterogeneous architectures. In fact, heterogeneous computing already dominates major market segments:

- **Heterogeneous Platforms in HPC**. Most multi-core processors for high-performance computing (HPC) are still homogeneous both in instruction set architecture and performance. They have a Thermal Design Power (TDP) of 100 Watts and integrate 4-16 heavyweight cores. However, as shown in Table 1.1, many of the most powerful supercomputers today (Top 500 list [144]) are based on platforms that combine multicore processors with data parallel accelerators. The fastest system, which is currenlty the Tianhe-2 supercomputer from China, uses Intel's Xeon Phi coprocessors and its runneup, which is the Titan supercomputer from the Oak Ridge National Laboratory, uses nVidia GPUs.

- **Heterogeneous processors in mobile platforms**. Heterogeneous architectures already provide power consumption advantages over homogeneous architectures. This is why they are extensively used today in low power embedded and mobile platforms. In this market segment processors integrate fewer cores and have lower TDP (around 2.5 to 10 Watts) than desktop and

high performance processors. However, they are able to achieve competitive performance integrating (1) data-parallel (graphics) accelerators with distinct programming and memory models [21, 57] and (2) many fixed-function accelerator blocks. Asymmetric "Big.LITTLE" architectures that combine different types of cores are also popular [74]. Since energy efficiency is of vital importance to have future Exascale system, some research projects [105] have envisioned the use of this kind of low-power heterogeneous processor on future high performance computing systems.

Position	Center	System	TFLOPS
1	**National Super Computer Center in Guangzhou, China**	Tianhe-2, Intel Xeon E5-2692 12C 2.2GHz, Intel Xeon Phi 31S1P	33,862.7
2	**Oak Ridge National Laboratory United States**	Titan (Cray XK7), AMD Opteron 6274 16C 2.2GHz, nVidia K20x	17,590.0
3	DOE/NNSA/LLNL United States	Sequoia - IBM BlueGene/Q, Power BQC 16C 1.60 GHz	17,173.2
4	RIKEN Japan	K computer - Fujitsu, SPARC64 VIIIfx 2.0GHz	10,510.0
5	DOE/SC/Argonne National Laboratory United States	Mira - IBM BlueGene/Q, Power BQC 16C 1.60GHz	8,586.6
6	**Swiss National Supercomputing Center Switzerland**	Piz Daint (Cray XC30), Intel Xeon E5-2670 8C 2.6Ghz, nVidia K20x	6,271.0
7	King Abdullah University Saudi Arabia	Shaheen II (Cray XC40), Intel Xeon E5-2698v3 16C 2.3Ghz,	5,537.0
8	**Texas Advanced Computing Center United States**	Stampede (PowerEdge C8220), Intel Xeon E5-2680 8C 2.7Ghz, Intel Xeon Phi SE10P	5,168.1
9	Forschungszentrum Juelich (FZJ) United States	JUQUEEN - IBM BlueGene/Q, Power BQC 16C 1.600GHz	5,008.9
10	DOE/NNSA/LLNL United States	Vulcan - IBM BlueGene/Q, Power BQC 16C 1.60GHz	4,293.3

Table 1.1: Five of the world's top 10 supercomputers in the latest edition of the TOP 500 list (June 2015) are based on heterogeneous systems that combine multicore processors and data parallel accelerators [144].

Figure 1.8 graphically illustrates the organization of an Intel-based heterogeneous server. This is is the building block of heterogeneous supercomputers and the kind of platform that we have investigated in this work. The hardware accelerator (or accelerators in multi-accelerator configurations) are attached directly via a fast PCI express link to the I/O HUB (IOH), which is already integrated on the processor die. Future system may integrate a single address space, but till now, the accelerators and the processor have independent memory spaces. Accelerators have a great deal of internal memory bandwidth but exchanging data between both spaces is a high latency operation that can cause huge bottlenecks. This forces programmers to design new algorithms that minimize memory transfers between the GPU and the host. When these transfers cannot be omitted, some code transformations can allow the overlapping of computation with memory transfers to hide such overheads.

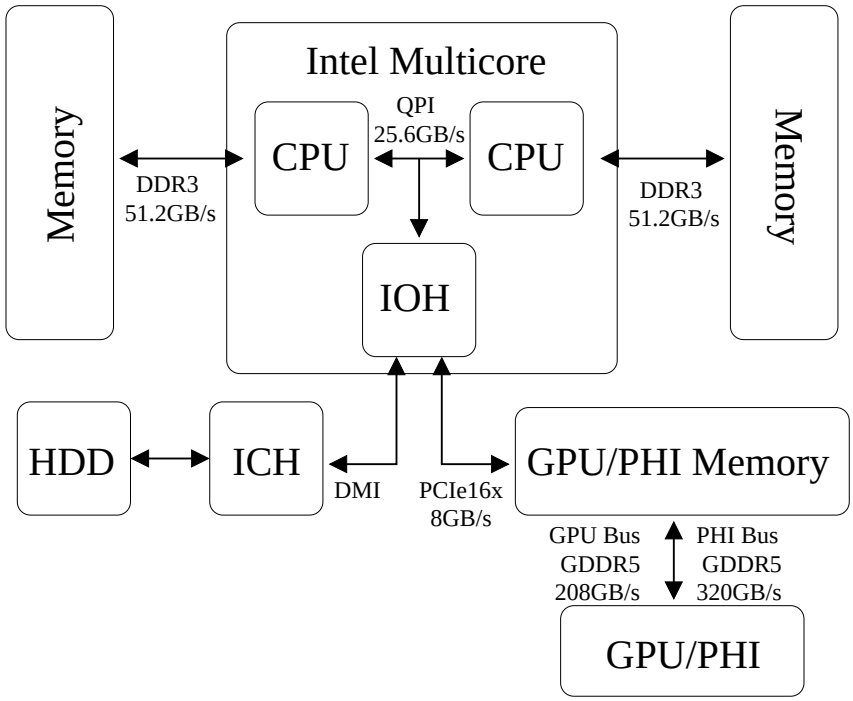


Figure 1.8: Intel-based (5520/5500 chipset) heterogeneous server [72].

Figure 1.9 shows an abstract block diagram of nVidia's (Kepler) GPU [155]. The GPU is organized into several multiprocessors, which in turn are composed of various simple processors (cores) that operates in SIMD fashion. The multiprocessors have fine grain multithreading capabilities, which means that they support hundreds of threads in-fly. Every multiprocessor switches to a different set of threads every clock cycle, which helps to maximize computational resources and hide the long latency memory accesses to a share GPU main memory.

The GPU main memory, usually called "global memory", is banked, which allows the hardware to coalesce several simultaneous memory accesses to adjacent positions into a single memory transaction. In addition, each multiprocessor contains a large set of registers and an on-chip SRAM scratchpad memory, i.e., a software controlled cache, to speed up data access. In more recent GPUs (starting from nVidia's Fermi architecture) the SRAM can be configured either as scratchpad or as cache memory and the user decide, with certain restrictions, the size of both memories. These newer GPUs also incorporate a L2 cache common to all multiprocessors. The access to the global memory can also be performed through special read-only two level hierarchy of so called texture caches, that are optimized to capture 2D access patterns [156].

Although GPUs are still the most popular data-parallel accelerator, we have also evaluated an Intel Xeon Phi device, another accelerator introduced by Intel to compete in this market. The Phi is a new family of processors based on the Intel MIC Architecture [75] that incorporates earlier work on the Larrabee architecture [124]. We have used the 22nm Knights Corner chip graphically described in Figures 1.10 and 1.11, which was the first commercial product from this family.

The Corner is a PCIe vector co-processor with integrates up to 61 in-order dual

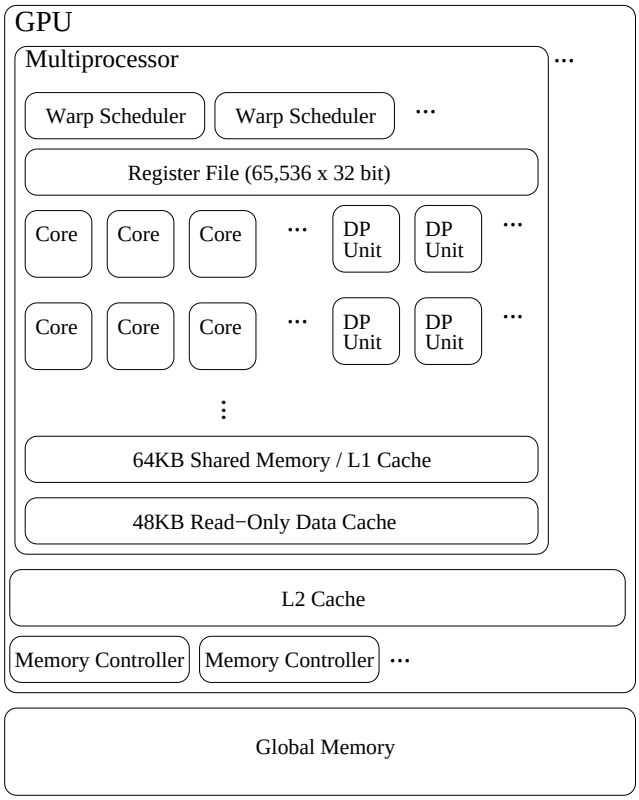


Figure 1.9: nVidia GPU (Kepler) architecture [155].

issue x86 cores, which trace some history to the original Pentium core, like the Larrabee predecessor. Among other enhancements, the Corner's cores are augmented with 64-bit support, 4 hardware threads per core (resulting in more than 200 hardware threads available on a single device) and 512-bit SIMD instructions [75]. Each core has a 512KB L2 cache locally but has also access to all other L2 caches in the system through a high-speed bidirectional ring [75]. Unlike previous GPUs, the L2 cache is kept fully coherent by a global-distributed tag directory.

The performance achieved by Knight Corner chips is usually outperformed by nVidia's counterparts [96]. However, last year Intel announced the Knight Landing processor [130] that should significantly improve MIC performance. The Landing

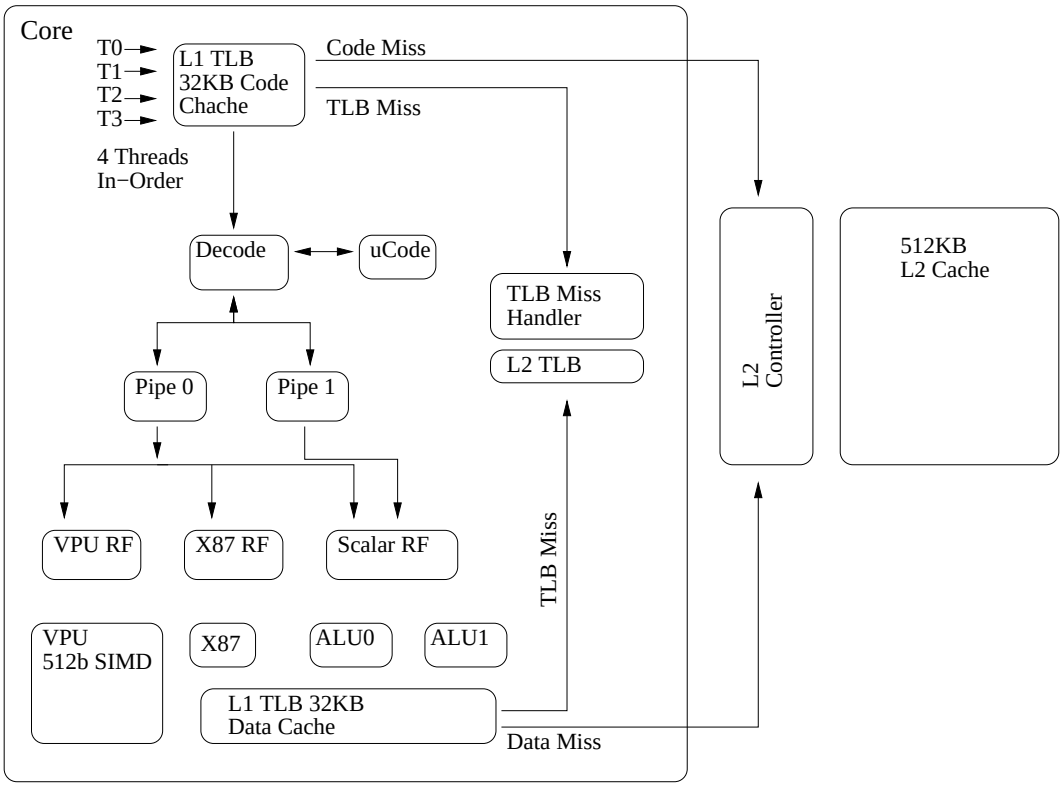


Figure 1.10: Architecture of a single Intel Xeon Phi Core [75].

is based on a different out-of-order Silvermont x86 core (used also by Atom processors) that implements the AVX-512 SIMD instructions [130]. Up to 72 core will be integrated on the same chip and the double precision floating point performance expected to exceed 3 TFLOPs [130]. Memory performance will also improve significantly thanks to the introduction of Micron's through-silicon vias (TSV)-based stacked DRAM [130]. The Landing main memory can scale up to 16GB of RAM while offering up to 500GB/sec of memory bandwidth, which is nearly 50% more than Knights Corner's GDDR5 [130]. From a systems perspective, the major change is that the Landing will be a standalone processor [130].

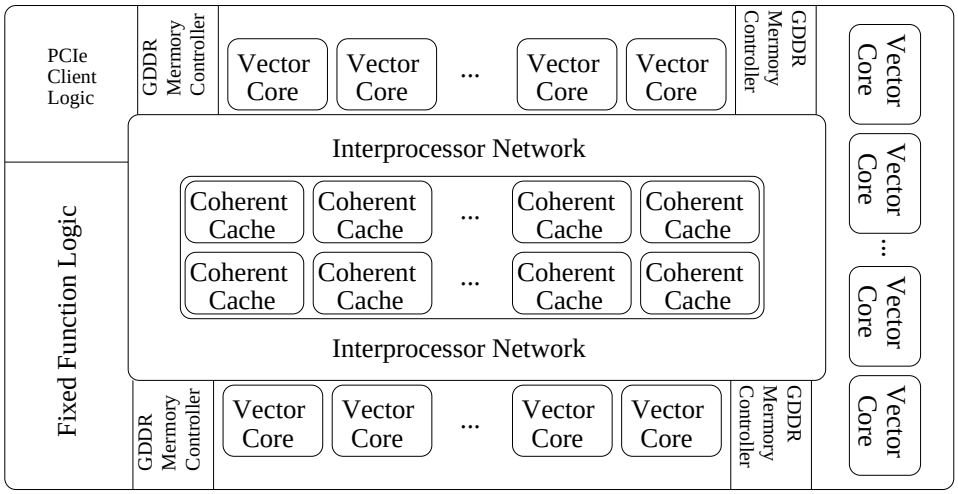


Figure 1.11: Micro-architecture of the Entire MIC coprocessor [75].

1.2.4. CUDA: a new language for many-core architectures

Early approaches to use the GPUs as a high performance data-parallel coprocessors were based on graphics frameworks such as OpenGL. With these graphics interfaces mapping applications into the GPU was a difficult task since programmers were forced to cast their computations in terms of graphics operations. Fortunately, in late 2006, nVidia introduced its CUDA programming model[2], which exposes application developers to an abstract model to make data parallel computing on a GPU more straightforward [161].

In this section we briefly describe the basics of this new language (see [118] for a detailed introduction). When programmed through CUDA, the GPU does not operates as a graphics pipeline, but as a data parallel coprocessor to the main processor or host. The programmer writes a serial host program, which runs on the host processor, that makes calls to data-parallel functions, known as CUDA kernels, that execute on the GPU in parallel. In addition, CUDA includes some extensions

[2]Originally, CUDA was the short for Compute Unified Device Architecture [94].

to explicitly manage data transfers from the host memory to the GPU main memory and vice versa.

A CUDA kernel executes in parallel across a set of parallel CUDA threads. Unlike regular threads, such as POSIX threads, CUDA threads are extremely lightweight and they have very little creation overhead. However, GPU needs thousands of threads for full efficiency. Typically, CUDA applications perform a sequence of kernels. Each kernel completes its execution before the next kernel begins [161]. In many applications, this is not a problem since kernels have enough parallelism to fill the entire GPU. Nevertheless, the latest GPUs have support form multiple independent kernels to execute simultaneously [161]. It is also possible to overlap kernel execution with data transfers between the host and GPU memories.

As shown in Figure 1.12, the programmer organizes CUDA threads into a hierarchy of grids of thread blocks [94]. Threads are grouped logically into CUDA blocks, and blocks are grouped into a CUDA grid. When invoking a kernel, the programmer specifies the number of threads per block and the number of blocks per grid [94]. As in other parallel programming languages, individual threads and blocks have different indices (thread and block IDs). These IDs are used to compute array subscripts when processing multidimensional data. The choice of the optimal size (dimension) of both CUDA grids and CUDA blocks, should be carefully chosen in order to achieve the maximum performance and unfortunately, it usually on the specific problem being treated.

Threads in a single CUDA block are executed on a single multiprocessor. This way, these threads can cooperate among themselves through barrier synchronization and shared access to the multiprocessor shared caches [94]. By contrast, the threads of different blocks in the same grid can only communicate through a high-latency

access to global memory. Note also that each multiprocessor can maintain hundreds of threads in execution.

Physically, these threads are organized in sets, called **warps**, the size of which is transparent to the programmer[3]. In every cycle, the hardware scheduler of each multiprocessor chooses the next warp to execute (i.e., no individual threads but warps are swapped in and out), using fine grain multithreading to hide memory access latencies. Threads within a warp are executed in lock step. If the threads in a warp execute different code paths due to a conditional branch, only those that follow the same path can be executed simultaneously and a penalty is incurred [156]. Therefore, warps should exhibit regular SIMD parallelism to avoid divergence. In addition, to minimize the number of non-coalesced memory accesses, threads in the same warp should access adjacent memory addresses.

In short, with this brief introduction we have tried to highlight that CUDA exposes many parameters to the programmer and those parameters really have a high impact on performance [28]. Furthermore, many of them are exclusively related to this specific type of architectures. Examples include explicit management of on-chip memory, aligned accesses to GPU memory, divergence control, correct adaptation to the SIMD programming paradigm, or convenient register usage, to name only a few. What is more important, many of these particularities usually vary from generation to generation of graphics processors. Thus, the programming effort invested to tune a particular routine for a specific GPU is usually not enough to tune the same code for a newer GPU. This is still, one of the main issues behind GPU programming.

[3]The warp size is related to the number of scalar processors that are in a multiprocessor. So far, the warp size of 32 has been kept constant across all CUDA-enabled GPU generations.

1.3. Related publications

The contributions presented in this thesis have been previously validated by several peer-reviewed publications in international conferences, and international journals. Next we list the publications directly related to this thesis.

I Pedro Valero-Lara, Alfredo Pinelli, Julien Favier and Manuel Prieto-Matías. Block Tridiagonal Solvers on Heterogeneous Architectures. IEEE 10th International Symposium on Parallel and Distributed Processing with Applications (ISPA), 609-616, 2012. doi: 10.1109/ISPA.2012.91.

II Pedro Valero-Lara, Alfredo Pinelli and Manuel Prieto-Matías. Fast finite difference Poisson solvers on heterogeneous architectures. Computer Physics Communications 185(4): 1265-1272, 2014. doi: 10.1016/j.cpc.2013.12.026.

III Pedro Valero-Lara, Alfredo Pinelli and Manuel Prieto-Matías. Accelerating Solid-fluid Interaction using Lattice-Boltzmann and Immersed Boundary Coupled Simulations on Heterogeneous Platforms. The International Conference on Computational Science (ICCS), 50-61, 2014.
doi: 10.1016/j.procs.2014.05.005.

IV Pedro Valero-Lara, Francisco D. Igual, Alfredo Pinelli, Manuel Prieto-Matías and Julien Favier. Accelerating fluid-solid simulations (Lattice-Boltzmann & Immersed-Boundary) on heterogeneous architectures. Journal of Computational Science, 2014. doi: 10.1016/j.jocs.2015.07.002.

The first paper (Chapter 2) describes our first attempts towards a parallel version of the *BLKTRI* [136] subroutine, limited to 2D problems. In the second paper

(Chapter 3), we extended that work to 3D problems using the FFT to uncouple a single 3D problem into a set of independent 2D problems. In the third paper (Chapter 4) we described our first attempt to design an efficient LBM-IB solver for heterogeneous multicore-GPU platforms. Finally, in the fourth paper (Chapter 5) we included additional optimizations, results on the Intel's Xeon Phi and a more elaborated discussion about performance results.

1.4. Conclusions

The overall goal of this work has been the design, implementation and evaluation of new parallel processing strategies to accelerate CFD problems with an efficient use of the non-homogeneous resources found on modern computing platforms.

Originally we focused on the acceleration of block tridiagonal solvers. These are one of the major bottlenecks in codes dealing with time-dependent elliptic partial differential equations, which is the target CFD problem under investigation. Related work have dealt with simpler scalar tridiagonal solvers. Based on them, in the first part of the thesis we explored and analysed different alternatives. The main conclusion that we found are the following:

- On multicore architectures, combining a coarse grain data distribution and the well-known Thomas algorithm is the best option, as in the scalar case.
- On GPUs, our fastest implementation is based on the PCR algorithm. For the scalar case, the latest work shows that recursive Doubling outperforms PCR. As an interesting note we need to highlight that we tried hard to validate that

results in our own codes using the information found in the related published papers. Unfortunately, all our attempts failed. We do not claim that PCR is the fastest method for the block case. It is possible that we are missing some optimization, but this clearly revealed the high complexity behind GPU code optimization.

- Our main contribution of this part of the thesis is the design an implementation of a hybrid solver. Essentially, we combine both the multicore and the GPU solver in a cooperative way that allow us to benefit from multicore-GPUs overlapping. Even for 3D problems with high arithmetic intensity, this combination is able to outperform homogeneous GPU implementations by a significant 15% margin.

Despite achieving significant speedups, the overall results does not meet the performance goals envisioned. In the second part of this thesis, we have tried to overcome this intrinsic limitation of Navier-Stokes solvers studying as an alternative the LBM method.

The design and implementation of parallel LBM solvers have been extensively studied. Several recent works have shown that the combination of hardware accelerators and methods based on LBM can achieve impressive performances due to the intrinsic characteristics of the algorithm. Certainly, the computing stages of LBM are amenable to fine grain parallelization in an almost straightforward way. We have confirmed these claims with our own implementation that includes most of the state-of-the-art code transformations that have been described in previous works.

But pure LBM solvers are not enough in many simulations. Indeed, our target simulator is an integrated framework that uses the Immersed Boundary (IB) method

to simulate the influence of a solid immersed in a incompressible flow. Unfortunately, when coupling LBM and IB methods, the overheads of IB correction degrade the overall performance. As an alternative, we have designed and evaluated hybrid implementations that effectively hide such overheads and allow us to exploit both the multi-core and the hardware accelerator in a cooperative way, with excellent performance results. Our experiments have revealed that for interesting physical scenarios, with realistic solid volume fractions, the proposed hybrid solvers are able to hide the overheads caused by the IB correction. Overall, we are able to match the performance of state-of-the-art pure LBM solvers on more complex simulations.

Notably, and this is one of the overall conclusion behind this work, we have used the same computing pattern to effectively accelerate the different CFD frameworks explored in this thesis. The main idea behind such pattern is to restructure the code to allow a better coordination between the host processor and the accelerator. With such coordination we have been able to hide (1) the cost of data transfers between the host and the accelerator main memories and (2) the overheads caused by sequential bottlenecks, i.e. the host processors is used as an accelerator of the sequential phases.

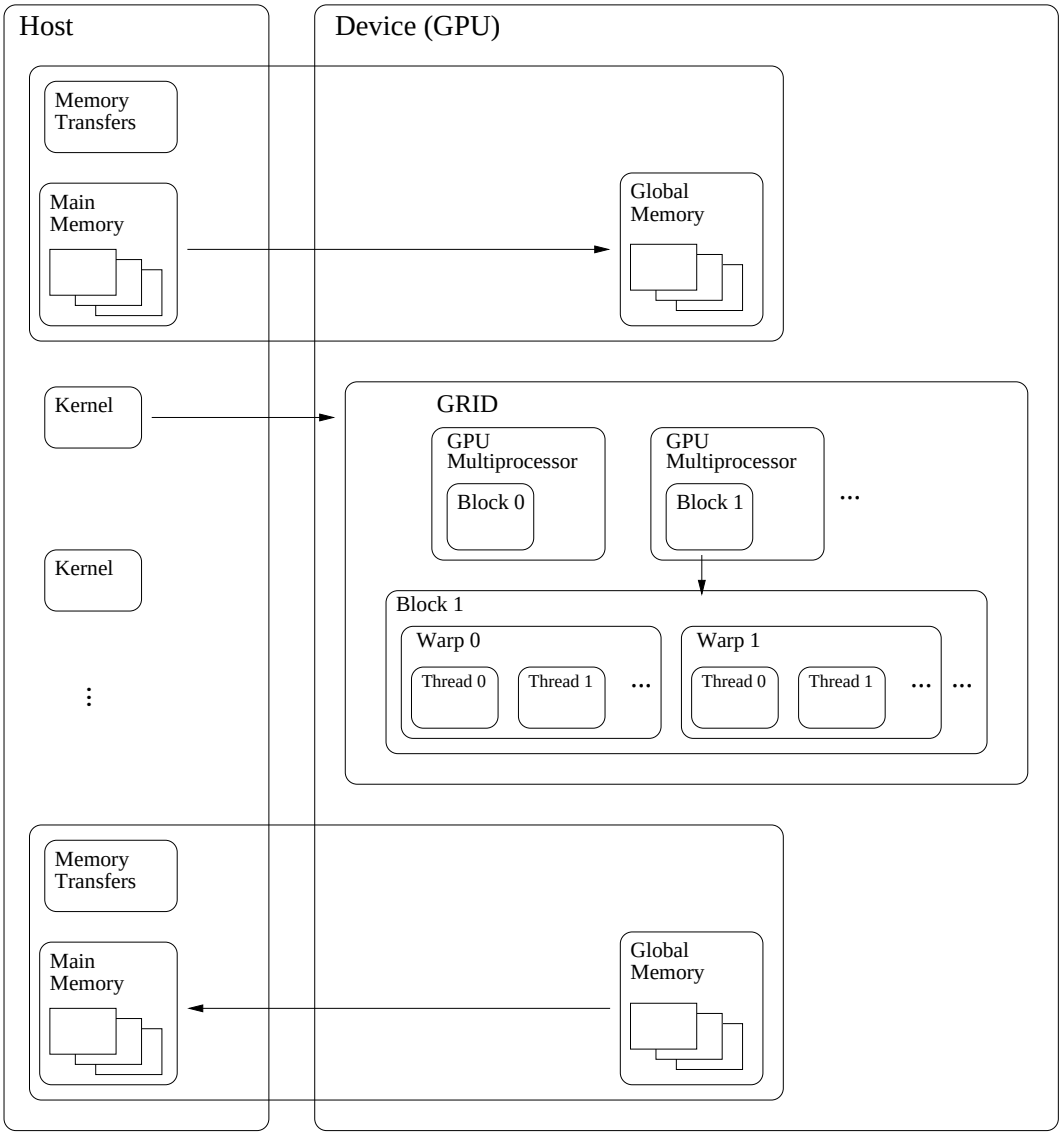


Figure 1.12: CUDA threads are arranged into groups, called CUDA blocks. A CUDA grid is a group of CUDA blocks.

Chapter 2

Block tridiagonal solvers on heterogeneous architectures

Modern multi-core and many-core systems offer a very impressive cost/performance ratio. In this work a set of new parallel implementations for the solution of linear systems with block-tridiagonal coefficient matrix on current parallel architectures is proposed and evaluated: one of them on multi-core, others on many-core and finally, a new heterogeneous implementation on both architectures. The results show a speedup higher than 6 on certain parts of the problem, being the heterogeneous implementation the fastest.

2.1. Introduction

The era of single-threaded processor has come to an end due to the limitation of the current Very Large Scale Integration (VLSI) technology. In response, most

hardware manufactures are designing and developing multicore processors and/or specialized hardware accelerators [52]. Programs will only increase in performance if they use and exploit the new parallel characteristics of new architectures. The recent appearance of GPUs for general purpose computing platforms offers powerful parallel processing capabilities at a low cost. On the other hand, current multicore processors are becoming another interesting parallel platform, for a low cost as well, and without the constraints of GPU architecture. Both types of processors offer a very interesting performance/cost ratio, and so they are increasingly used in parallel computing.

In this work, a study on the parallel characteristics of an algorithm for the direct solution of linear systems with a block-tridiagonal coefficient matrix (*BLK-TRI* problem) is carried out, by exploiting current parallel architectures including heterogeneous (i.e., multi and many cores) ones. The efficient resolution of block tridiagonal linear systems is of fundamental importance in computational mechanics since they stem from classical finite difference discretizations of two dimensional separable elliptic equations. In particular, the most time consuming part of almost any incompressible Navier Stokes solver (i.e., incompressible fluid dynamic simulation codes) is related to the solution at each time step of a *pressure Poisson* equation, which leads to a block tridiagonal linear system after classical finite differences discretization.(see for instance [58]). The achievement of a satisfactory computational efficiency to tackle this class of problems is therefore a key issue when simulating unsteady fluid flow processes (turbulent flows for instance). Although, there is no previous work that provides a detailed study and/or proposes a new parallel algorithm on current parallel architectures to deal with linear systems characterized by a block tridiagonal coefficient matrix. However, in literature other

authors address topics which are somehow related to the present contribution. In particular, the core subject of our work is proposed as a future research line in Y. Zhang et al. [166], which introduces and evaluates several methods to solve tridiagonal systems on GPUs. Also, D. Göddeke et al. [53] propose to use Cyclic Reduction method to solve simultaneously several tridiagonal systems on GPUs, a computational problem which arises when dealing with a line relaxation type multi-grid solver applied to elliptic partial differential equations discretized on structured grids. However, both works do not cover neither the direct solution of the block tridiagonal case, nor explore the possibility of combining multi and many-core architectures.

This work is structured as follows. Section 3.3 introduces the problem we wish to tackle: efficient and direct solution of block-tridiagonal linear systems of equations. In Section 5.6.4.1 the particular characteristics of current many-core architectures are briefly recalled. Section 2.4 presents the parallel algorithms we have considered to solve block-tridiagonal problems, and in Section 2.5 different parallel implementations are proposed for them. Section 2.6 contains a performance analysis, of the proposed techniques and finally, in Section 3.7 some conclusions and directions for future work are outlined.

2.2. The block tridiagonal system algorithm

In this section we briefly summarize a classical direct method for the discrete solution of separable elliptic equations based on a block cyclic reduction algorithm (i.e., *BLKTRI* routine in the *fishpack* package available at *netlib*) [136]). This method is commonly used when tackling the solution of a linear system of equations

arising from the second order centered finite difference discretization of 2D separable elliptic equations. From the standpoint of computational complexity, (speed and storage), for a $m \times n$ net, its operation count is proportional to $mn\log_2 n$, and the storage requirements are minimal, since the solution is returned in the storage occupied by the right side of the equation (i.e., $m \times n$ locations are required). More in details, consider the 2D separable elliptic equation having $x(u,v)$ as unknown field:

$$\frac{\partial}{\partial u}\left(a(u)\frac{\partial x}{\partial u}\right) + b(u)\frac{\partial x}{\partial u} + c(u)u + \frac{\partial}{\partial v}\left(d(v)\frac{\partial x}{\partial v}\right) + e(v)\frac{\partial x}{\partial v} + f(v)u = g(u,v) \tag{2.1}$$

If we discretize (3.6) with given Dirichlet or Neumann boundary conditions assigned on the edges of a square, using the usual five-point scheme with the discrete variables ordered in a lexicographic fashion, we obtain a linear system of $m \times n$ equations (having m nodes in the u direction and n in v): $\mathbf{A}\tilde{\mathbf{x}} = \tilde{\mathbf{g}}$, where $\mathbf{A}$ is a block tridiagonal matrix:

$$\mathbf{A} = \begin{bmatrix} B_1 & C_1 & & & & 0 \\ A_2 & B_2 & C_2 & & & \\ & \cdot & \cdot & \cdot & & \\ & & \cdot & \cdot & \cdot & \\ & & & A_{n-1} & B_{n-1} & C_{n-1} \\ & & & & A_n & B_n \end{bmatrix}$$

and the vectors $\vec{x}$ and $\vec{g}$ are consistently split as a set of sub-vectors $\vec{x}_i$ and $\vec{g}_i$, $i =$

1..m, of length m each:

$$\mathbf{x} = [\tilde{\mathbf{x_1}},\ \tilde{\mathbf{x_2}}, \cdots\cdots \tilde{\mathbf{x_n}}]^{\mathbf{T}}$$

$$\mathbf{g} = [\tilde{\mathbf{y_1}},\ \tilde{\mathbf{y_2}}, \cdots\cdots \tilde{\mathbf{y_n}}]^{\mathbf{T}}$$

There is no restriction on m; however, cyclic reduction algorithms require $n = 2^k$, with large values of k for optimal performances. The blocks $\mathbf{A_i}$, $\mathbf{B_i}$ and $\mathbf{C_i}$ are $m \times m$ square matrices. In particular, the *BLKTRI* algorithm requires them to be of the form:

$$A_i = a_i I \tag{2.2}$$

$$B_i = B + b_i I \tag{2.3}$$

$$C_i = c_i I \tag{2.4}$$

where a_i, b_i and c_i are scalars. Having used a standard five point stencil for the discretization of (3.6), the matrix B is of tridiagonal pattern. The solution is obtained using an *extended cyclic reduction algorithm* which consists of the following phases:

1. Preprocessing phase: a set of intermediate results that only depend on the entries of **A** (not on the right hand side *rhs* of the equation) are obtained. Those results may be stored if a number of linear systems sharing the same coefficient matrix with different *rhs* $\vec{g}$ need to be solved.

2. Reduction phase: A sequence of linear system is generated starting from the original complete one by decoupling odd and even equations. At each step about half the unknown vector $\vec{x_i}$ are eliminated with the result that

each system has a block order of about half the former one. This process is continued until a system with the single unknown vector $\vec{x_2^k}$ is obtained.

3. Back-substitution phase: The solution vectors $\vec{x_i}$ are determined by first solving the final system generated in the above phase $\vec{x_2^k}$. Then the linear systems are solved in reverse order determining more $\vec{x_i}$ solution vectors, using those $\vec{x_i}$ previously computed.

During the reduction phase the following equations are tackled and solved [136]:

$$q_i^{(r)} = (B_i^r)^{-1} B_{i-2^{r-1}}^{r-1} B_{i+2^{r-1}}^{r-1} p_i^{(r)} \tag{2.5}$$

$$p_i^{(r+1)} = \alpha_i^r (B_{i-2^{r-1}}^{r-1})^{-1} q_{i-2^r}^{(r)} + \gamma_i^r (B_{i+2^{r-1}}^{r-1})^{-1} q_{i+2^r}^{(r)} - p_i^r \tag{2.6}$$

where **g** is split in two different terms, q and p. B stores the roots calculated in preprocessing phase. This procedure is required to stabilize the method [136]. α and γ have the following form:

$$\alpha_i^{(r)} = \prod_{j=i-2^{r+1}}^{i} aj \tag{2.7}$$

$$\gamma_i^{(r)} = \prod_{j=1}^{i+2^{r-1}} cj \tag{2.8}$$

Conversely, in the last phase the following equations are solved [136]:
for $r = k, k-1, \ldots, 0$ and $i = 2^r, 3\times 2^r, 5\times 2^r, \ldots, 2^{k-r}\times 2^r$:

$$x_i = (B_i^r)^{-1} B_{i-2^{r-1}}^{r-1} B_{i+2^{r-1}}^{r-1} [p_i^{(r)} - \alpha_i^r (B_{i-2^{r-1}}^{r-1})^{-1} x_{i-2^r} - \gamma_i^r (B_{i+2^{r-1}}^{r-1})^{-1} x_{i+2^r}] \tag{2.9}$$

This method is implemented in a FORTRAN package library called FISHPACK, which is widely used and well known within the computational fluid dynamics community [48].

2.3. GPU (many-cores architecture)

Although GPUs are traditionally associated to interactive applications involving high rasterization performance, they are also widely used to accelerate much more general applications (now called General Purpose Computing on GPU (GPGPU) [106]) which require an intense computational load and present parallel characteristics.

The main feature of these devices is a large number of processing elements integrated into a single chip, which reduces significantly the cache memory. These processing elements can access to a local high-speed external DRAM memory, connected to the computer through a high-speed I/O interface (PCI-Express).

Overall, these devices can offer a higher main memory bandwidth and can use data parallelism to achieve a higher floating point throughput than CPUs [44].

Figure 2.1 (left) describes the architecture of modern NVIDIA's GPUs. It consists of a number of multiprocessors and each multiprocessor has a set of simple cores. All multiprocessors share the same main memory, called "global memory". In addition, all cores of one multiprocessor can access to the same "shared memory". This memory is useful when many threads have to access to the same data or if one data is used many times by one thread. Indeed when a block of information has to be loaded in shared memory it is necessary to take it from global memory.

To control the GPU devices and manage memory, we have used in the present work the high level programming language CUDA [28], introduced by NVIDIA.

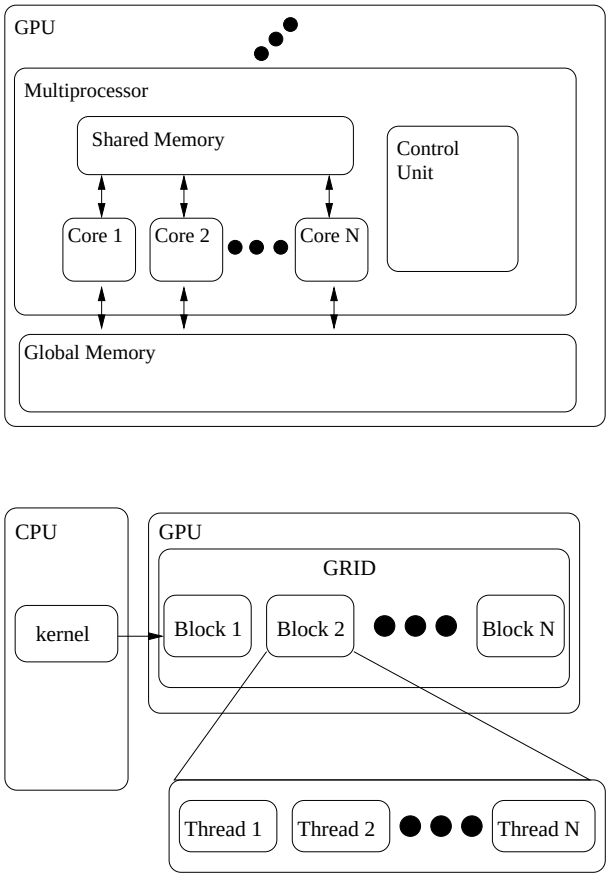


Figure 2.1: GPU architecture (top) and grid of CUDA blocks (bottom).

Calculations in CUDA are distributed into a mesh or grid of thread blocks of the same size (number of threads). These threads run the GPU code, known as kernel; note that although this kernel is originally called by the CPU, finally it is executed in the GPU, as seen in Figure 2.1 (right). Threads within a blocks are grouped into warps of 32 threads. A warp executes one common instruction at a time, so full efficiency is realized when all 32 threads of a warp agree on their execution path. If threads of a warp diverge via a data-dependent conditional branch, the warp serially executes each branch path taken, disabling threads that are not on that path, and when all paths complete, the threads converge back to the same

execution path. Branch divergence occurs only within a warp; different warps execute independently regardless of whether they are executing common or disjoint code paths. The dimensions of both the mesh and the threads block should be carefully chosen in order to achieve the maximum performance depending on the specific problem being treated.

The threads within a block can work together efficiently exchanging data via a local shared memory and synchronize low-latency execution through synchronization barriers (where threads in a block are suspended until they all reach the synchronization point). By contrast, the threads of different blocks in the same network can only communicate through a high-latency access to global memory (the memory graphic board). In order to exploit the bandwidth of both global and shared memory in an efficient way, it is better that threads have the same or very similar pattern of memory access to reach contiguous spaces of memory (coalescing access). Besides, another technique used to avoid the latency of the global memory, consists in overlapping the executions of threads blocks with accesses to global memory.

All CUDA code is divided in two different parts, CPU code and GPU code. The CPU code, provides the instructions to be performed by the CPU, e.g. allocating data on the CPU and GPU, transferring data between GPU and CPU and launching kernels. On the other hand, the GPU code (kernel) provides the instructions to be executed in the GPU, by all threads of the kernel.

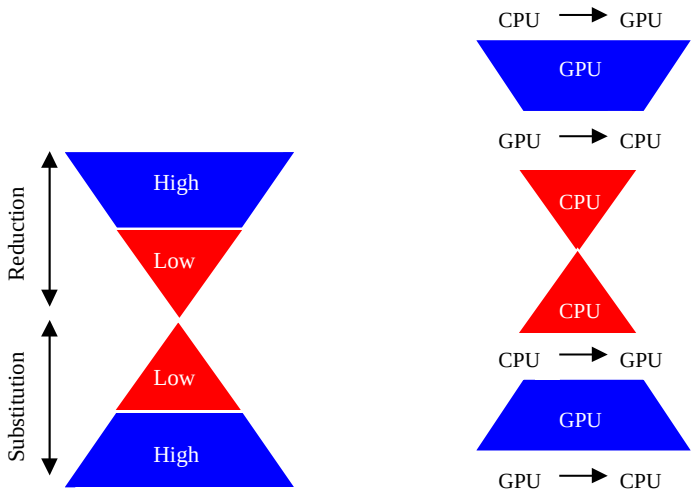


Figure 2.2: Spaces of parallelism (left) and heterogeneous implementation steps (right).

2.4. Parallel block tridiagonal solver

This section presents the parallel strategy adopted to improve the BLKTRI algorithm performance.

We start by noting that the reduction and substitution phases of the algorithm involve the use of the cyclic reduction method, and that all elements of each step are independent; therefore they can be calculated simultaneously. The level of parallelism is divided by 2 step by step during the reduction phase, and multiplied by 2 during the substitution phase.

Moreover, in each step of both phases, the algorithm displays other features amenable to parallelism:

For $s = 1, 2, \ldots, k$ steps, we have 2^{k-s} independent terms in the reduction phase and 2^{s-1} independent terms in the substitution phase. Firstly, during the reduction phase, it is necessary to compute the $q^{(r)}$ terms; after this stage, the terms $\alpha_i^r (B_{i-2^{r-1}}^{r-1})^{-1} q_{i-2^r}^{(r)}$ and $\gamma_i^r (B_{i+2^{r-1}}^{r-1})^{-1} q_{i+2^r}^{(r)}$ of equation 3.11 can be computed si-

multaneously, since they are independent (having processed $q^{(r)}$ already). Finally, we just need to sum those terms to obtain $p^{(r)}$.

Parallel features of the same kind can be observed in the substitution phase which is carried out in a similar way. First, $\alpha_i^r(B_{i-2^{r-1}}^{r-1})^{-1}x_{i-2^r}$ and $\gamma_i^r(B_{i+2^{r-1}}^{r-1})^{-1}x_{i+2^r}$ are calculated according to equation 3.14. As in the reduction phase, those are independent and therefore can be computed simultaneously. The obtained results are added to $p^{(r)}$, to obtain the (β_i) terms by tacking $(B_i^r)^{-1}B_{i-2^{r-1}}^{r-1}B_{i+2^{r-1}}^{r-1}\beta_i$. Finally, x_i is obtained.

The described sequence can be summarized as a reduction phase going through the following stages: *i)* compute $q^{(r)}$, *ii)* determine $\alpha^{(r)}$ and $\gamma^{(r)}$, and *iii)* finally $p^{(r)}$; and a substitution phase, where β is first computed to achieve the final solution vector x.

To obtain the aforementioned terms the solution of a set of tridiagonal systems of equations must be faced. The solution of these systems represent the most expensive stage of the algorithm. Also, other more basic mathematical operations such as vectors sums or scalar vector multiplications introduce a non negligible cost. The original sequential implementation of the algorithm in the *fishpack* package makes use of the Thomas algorithm [121] to tackle the solution of each tridiagonal problem. Even if Thomas algorithm is a very efficient sequential method, there exist other algorithms to solve tridiagonal methods that present better features under the parallel point of view. Based on previous works [166, 53], we have focused on the Cyclic Reduction (CR) and the Parallel Cyclic Reduction (PCR) algorithms. Using parallel versions of those methods, that will be described later on, the parallelism of the tridiagonal algorithm is multiplied by m, (m being the size of vectors p, q and x in equations 3.9, 3.10 and 3.14).

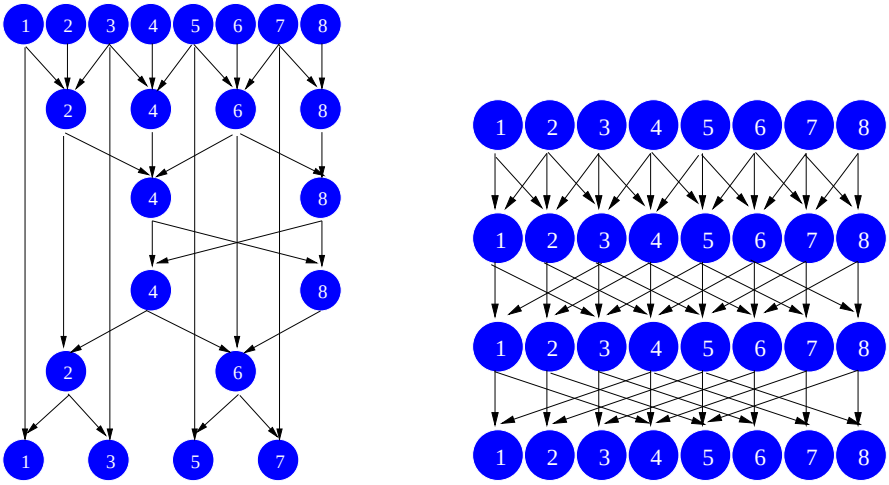


Figure 2.3: Pattern of communications for CR (left) and PCR (right) algorithm.

Figure 3.9 (left) schematically shows the potential for parallelism of the problem following the sequence of the algorithm. In the figure, we have highlighted three different stages: two with a high level of intrinsic parallelism and one with a lower potential one. The two highly parallel ranges correspond to the first and last steps of the reduction and substitution phase, respectively. The red area refers to low parallelism corresponding to the last steps of the reduction phase and the first steps of the substitution phases.

2.5. Parallel implementation

In order to exploit the parallel potential features presented in the previous section, we have implemented a set of different algorithms on current shared memory parallel architectures with multi and many-cores using Open-MP and CUDA software libraries. Since the core of the algorithm is based on the parallel solution of tridiagonal systems, first we will recall the basic features of the problem at hand. A

tridiagonal system has the form $Ax = y$, where A is a tridiagonal matrix

$$A = \begin{bmatrix} b_1 & c_1 & & & & 0 \\ a_2 & b_2 & c_2 & & & \\ & . & . & . & & \\ & & . & . & . & \\ & & & a_{n-1} & b_{n-1} & c_{n-1} \\ & & & & a_n & b_n \end{bmatrix}$$

The classical method to tackle this problem is the Thomas [121] algorithm, which is a Gaussian elimination tailored to the tridiagonal matrix case. The algorithm relies on two stages: forward elimination and backward substitution. The forward elimination consists in:

$$c'_1 = \frac{c_1}{b_1}, c'_i = \frac{c_i}{b_i - c'_{i-1} a_i}, i = 2, 3, \ldots, n-1$$
$$y'_1 = \frac{y_1}{b_1}, y'_i = \frac{y_i - y'_{i-1} a_i}{b_i - c'_{i-1} a_i}, i = 2, 3, \ldots, n-1$$

And the backward stage reads:

$$x_n = y'_n, x_i = y'_i - c'_i x_{i+1}, i = n-1, n-2, \ldots, 1$$

Therefore, the complexity of Thomas algorithm is of $8n$ operations and requires $2n$ steps.

On a multi-core platform the parallel implementation consists in distributing the elements of the terms described in Section 3.3 on different cores by using Open-MP pragmas. Blocks of continuous systems are assigned to each thread, allowing in this way an efficient use of shared memory and the parallel resources of many-core architecture.

Concerning the many-core architecture, three different implementations have been analyzed. The first one, is similar to the multi-core counterpart and maps tridiagonal systems to threads, i.e. each thread blocks solves a set of independent systems that are solved by using the Thomas algorithm. Conversely, the other two methods exploit a finer grain parallelism by mapping each tridiagonal system onto a thread block to solve tridiagonal systems while taking care of basic operations on vectors within the single tridiagonal algorithm too.

In the following, we give more details on the way we have introduced parallelism to solve many tridiagonal systems in the above mentioned spirit.

- Cyclic Reduction (CR) [69]. This method is divided in two phases, reduction and substitution. For the sake of simplicity, in the next explanation, we will skip the description of the special treatment of the last step of reduction phase and of the first step of the substitution phase. In each intermediate step of the reduction phase, all even-indexed (i) equations $a_i x_{i-1} + b_i x_i + c_i x_{i+1} = d_i$ are dealt with in parallel. The values of a_i, b_i, c_i and d_i are updated in each step according to:

$$a'_i = -a_{i-1}k_1, b'_i = b_i - c_{i-1}k_1 - a_{i+1}k_2$$
$$c'_i = -c_{i+1}k_2, y'_i = y_i - y_{i-1}k_1 - y_{i+1}k_2$$

$$k_1 = \frac{a_i}{b_{i-1}}, k_2 = \frac{c_i}{b_{i+1}}$$

  All odd-indexed unknowns x_i are solved in the substitution phase by introducing the already achieved x_{i-1} and x_{i+1} values:

$$x_i = \frac{y'_i - a'_i x_{i-1} - c'_i x_{i+1}}{b'_i}$$

This algorithm needs $17n$ operations and $2\log_2 n - 1$ steps and its communication pattern is shown in Figure 2.3 (left).

- Parallel Cyclic Reduction (PCR) [70] is a variant of CR, which only has substitution phase. For convenience, we consider cases where $n = 2^s$, that involve $s = \log_2 n$ number of steps. Similarly to the former algorithm, a, b, c and y are updated as follows, for $j = 1, 2, \ldots, s$ and $k = 2^{j-1}$:

$$a'_i = \alpha_i a_i, b'_i = b_i + \alpha_i c_{i-k} + \beta_i a_{i+k}$$
$$c'_i = \beta_i c_{i+1}, y'_i = b_i + \alpha_i y_{i-k} + \beta_i y_{i+k}$$
$$\alpha_i = \frac{-a_i}{b_{i-1}}, \beta_i = \frac{-c_i}{b_i}$$

  finally the solution is achieved as:

$$x_i = \frac{y'_i}{b_i}$$

  The operation count of the algorithm is $12n\log_2 n$. The corresponding communication pattern is sketched in Figure 2.3 (right).

For both CR and PCR algorithms, the terms a, b, c and y are stored in shared memory, since they are shared by threads of the same block. From the standpoint of GPU computing, the differences between these two algorithms have an important impact on the respective performances. Indeed, in order to respect data dependencies and to avoid Read-After-Write (RAW) risks, it is necessary to introduce additional synchronizations points between threads. In the CR algorithm synchronizations are introduced at the end of each step. In the PCR algorithm an additional synchronization point is necessary, since threads read elements, which are written

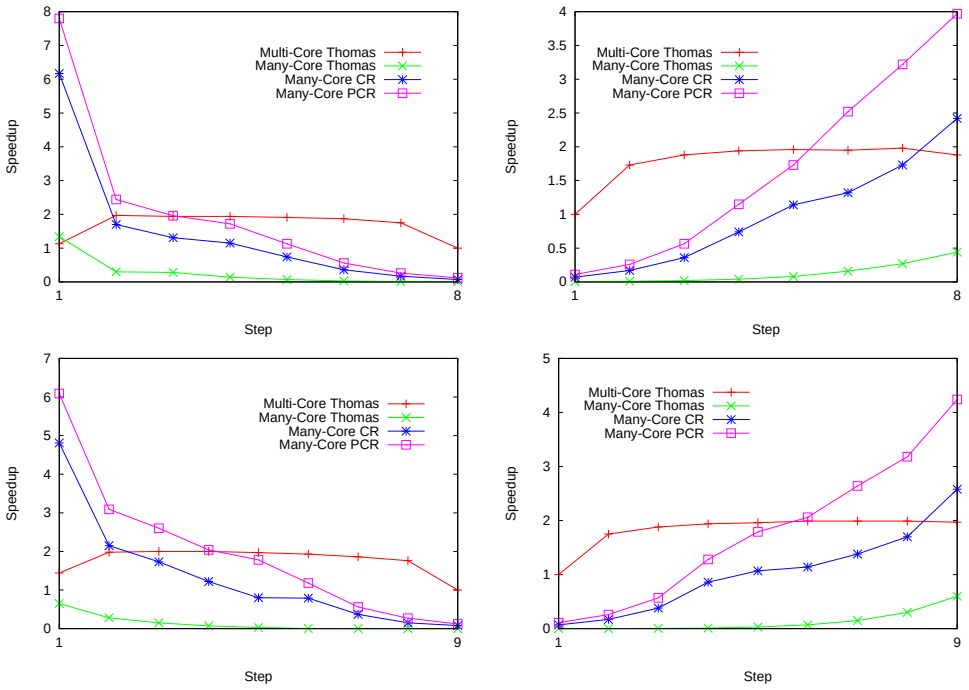


Figure 2.4: Speedup obtained in each step of the reduction (left) and substitution (right) phases, for 256×256 (top) and 512×512 (bottom) problems.

by other threads in the same step. This extra synchronization might be alleviated by the fact that PCR needs less steps than CR. Moreover, the two algorithms exhibit different memory access pattern (as shown in Figure 2.3), since all accesses done by PCR algorithm are coalesced, whereas they are not in the case of the CR algorithm.

Next, we introduce a final heterogeneous algorithm, which is a combination of the multi and many-core procedures discussed above, thus able to exploit the characteristics of both architectures. We notice that the potential for parallelism of our problem is dynamic, i.e., increases and decreases over the algorithmical sequence. As already mentioned in section 5.6.4.1, applications executed on a many-core device must has parallel characteristics and an intense computational load to obtain good performances. Thus, only the first steps of the reduction phase

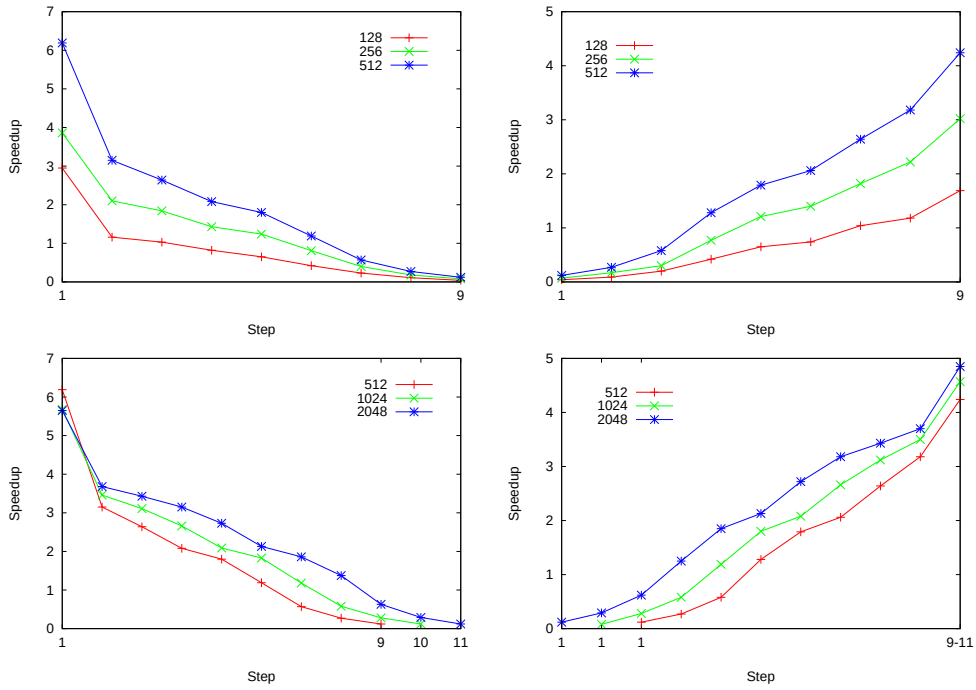


Figure 2.5: Trend of speedup in reduction (left) and substitution (right) phases, increasing the size of both, m (top) for n equal to 512, and n (bottom) for m equal to 512.

and the last steps of the substitution phase, which have the highest computational load can be effectively accelerated by using many-core architecture; in the other operational range of the algorithm multi-core architecture is preferred. This idea is illustrated in Figure 3.9 (right): at the beginning of the initial stage the transfer from the main memory to global memory of arrays a, b, c, e, f and g (equation 3.6) is required; for all the other stages only the transfer of g is needed.

2.6. Performance evaluation

In this section a performance analysis is carried out considering the sequential implementation of BLKTRI included in FISHPACK, and the different parallel

implementations using current multi-core and many-core architectures previously presented. All the results are given in terms of speedup and execution time using an Intel Xeon E5520 processor, and a Nvidia Tesla C1060 GPU platform (see Table 2.1). Due to limitation of our GPU platform, the maximum size of threads block is limited to 512, which poses a maximum number of rows that can be considered in the problem.

Platform	Xeon E5520 (2.26 GHz)	Tesla C1060
Multiprocessors (MP)	0	30
Cores	4	240
on-chip Memory	L1 32KB (per core) L2 256KB (unified) L3 8MB (unified)	16KB (per MP)
Memory	16GB DDR3	4GB GDDR3
Bandwidth	25.6 GB/s	102 GB/s
OS	Linux Ubuntu 10.10 amd-64	
Compiler	The Portland Group (PGI) Fortran	
	$-mp$	$-Mcuda$

Table 2.1: Platforms.

Figure 3.10 shows the speedup achieved in each step for both phases (i.e., reduction and substitution) considering all the implementations presented above. The Open-MP implementation (multi-core Thomas) shows a speedup of around 1.9 by using 4 cores.

The three implementations on the many-core platform achieved different results. The many-core Thomas version is found to be the slowest, since it only exploits coarse-grained parallelism (parallelism between tridiagonal systems). Thus this method is more suitable to a multi-core architecture. The other two versions (many-core CR and many-Core PCR) provide better results, since they exploit fine grained parallelism at tridiagonal solvers level. The better performances (speedup reaching

a value of 7.8) are obtained in the first steps of reduction phase and in the last steps of the substitution phase because of the highest level of inherent parallelism. Although PCR is more expensive computationally than CR, we note that it is able to outperform the latter since it provides a more efficient memory access pattern, allowing for both coalesced accesses, and a better exploitation of the shared memory. More in general, is is observed that shared memory improves performance of PCR and CR by 20% and 3% respectively. Figure 2.5 shows the trend of speedup when using the best many-core version while increasing m and n (i.e., m and n be the numbers of rows and columns of the problem). The two first graphs on the top refer to $n = 512$ and $m = 128, 256$ and 512, and the bottom graphs refer to $m = 512$ and $n = 512$, 1024 and 2048 respectively. For both cases, it is clear that speedup improves when the size of the problem is increased.

Also note that, although the approach to parallelism is different in each stage, the computational load is kept very similar, since at each step the number of tridiagonal systems to be solved is increased. This feature is illustrated in Figure 2.6 that shows the execution time, step by step, for both phases.

As a final comparison, Figure 5.19 shows the execution time required for the sequential, multi-core and heterogeneous implementations. For the parallel implementations the execution time includes the transfers between CPU and GPU as well (Heterogenous*). In particular, as far as transfer timings are concerned, it is noted that when increasing the size of the problem, the percentage of time related to transfers decreases: 3.29% and 3.05% for cases 256×256 and 256×512, and 3.04% and 2.12% for 512×512 and 512×1024 respectively.

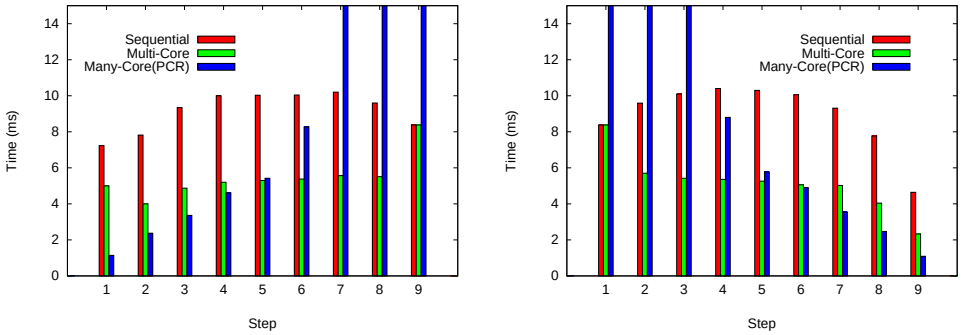


Figure 2.6: Execution time step by step for both phases, reduction (left) and substitution (right) for a 512×512 problem.

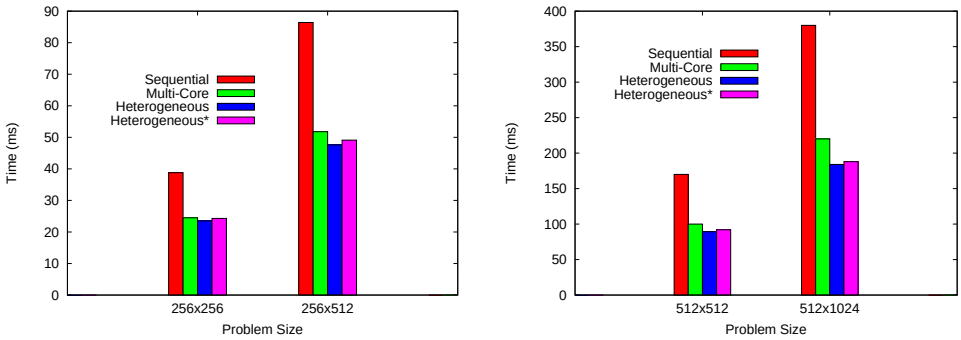


Figure 2.7: Total execution time.

2.7. Concluding remarks

In this chapter, we have presented different parallel approaches to tackle the solution of block tridiagonal linear systems considering the various possibilities offered by currently available architectures. In particular, we have focused on parallelization issues related to the *BLKTRI* routine of the *Fishpack* library. The performances of each parallel implementation proposed have been measured in terms of speedup and execution time to select the most efficient approach for this class of problem. The efficient solution of such linear systems is indeed of crucial importance for being the major bottleneck of several large scale simulation codes

dealing with time-dependent elliptic partial differential equations discretized using finite differences.

The most efficient implementation turned out to be a hybrid implementation where the stages of the algorithm presenting the highest potential for fine grained parallelism are computed on the GPU while the rest is assigned to the multi core processor. The former is a set of large tridiagonal systems that are solved with a PCR algorithm, whereas the latter is another set of (smaller) tridiagonal problems that are solved sequentially using the Thomas algorithm. For large enough problem sizes, the overhead caused by transfers between both architectures become negligible.

Based on these insights, in the next chapter we generalize our codes to three-dimensional problems with periodic boundary conditions in one direction. These problems can be efficiently transformed into a set of independent block tridiagonal problems using a *FFT* transform. Therefore, our three-dimensional solver is also based on the main algorithms described above.

Chapter 3

Fast finite difference poisson solvers on heterogeneous architectures

In this chapter we propose and evaluate a set of new strategies for the solution of three dimensional separable elliptic problems on CPU-GPU platforms. The numerical solution of the system of linear equations arising when discretizing those operators often represents the most time consuming part of larger simulation codes tackling a variety of physical situations. Incompressible fluid flows, electromagnetic problems, heat transfer and solid mechanic simulations are just a few examples of application areas that require efficient solution strategies for this class of problems. GPU computing has emerged as an attractive alternative to conventional CPUs for many scientific applications. High speedups over CPU implementations have been reported and this trend is expected to continue in the future with improved programming support and tighter CPU-GPU integration. These speedups by no means imply that CPU performance is no longer critical. The conventional CPU-control-GPU-compute pattern used in many applications wastes much of CPU's

computational power. Our proposed parallel implementation of a classical cyclic reduction algorithm to tackle the large linear systems arising from the discretized form of the elliptic problem at hand, schedules computing on both the GPU and the CPUs in a cooperative way. The experimental result demonstrates the effectiveness of this approach.

3.1. Introduction

The era of single-threaded processors has come to an end due to the limitation of the CMOS technology and in response, most hardware manufactures are designing and developing multi-core processors and specialized hardware accelerators such as GPUs [52, 22, 93]. As a consequence, applications can only improve their performance if they are able to exploit the available parallelism of the new architectures.

In this chapter we study the implementation of a fast solver based on a block cyclic reduction algorithm to tackle the linear systems that arise when discretizing a three dimensional separable elliptic problem with standard finite difference. A clear example of the importance of dealing efficiently with three dimensional elliptic systems is found in the numerical simulation of incompressible fluid flows. Indeed, the most time consuming part of almost any incompressible unsteady Navier Stokes solver (i.e., incompressible fluid dynamic simulation codes) is related to the solution of a *pressure Poisson* equation at each time step (see for instance [58]). The achievement of a satisfactory computational efficiency to tackle this class of elliptic partial differential equations is therefore a key issue when simulating unsteady fluid flow processes (turbulent flows for instance).

Other authors have addressed topics which are somehow related to the present contribution. [133] analyzes the performance of a block tridiagonal benchmark on GPUs. This is the first known implementation of a block tridiagonal solver in CUDA but the pattern of the block matrices they analyzed differ from our target problem. The sub-matrix element rank (m) was assumed to be small ($m = 5$). In our case both m and the arithmetic intensity of problem are higher.

For distributed multicore clusters, the BCYCLIC algorithm developed by Hirshman et al. [68] is able to solve linear problems with dense tridiagonal blocks. Our target algorithm, the BLKTRI code [136] is not well-suited for dense blocks but it is the most popular approach for solving block tridiagonal matrices which arise from separable elliptic partial differential equations.

Many authors have studied the implementation of scalar tridiagonal solver on GPUs [53, 166, 116, 34, 80]. D. Göddeke et al. [53] proposed an efficient implementation of the Cyclic Reduction (CR) algorithm, which is used as a line smoother in a multigrid solver. Yao Zhang et al. [166] proposed some hybrid algorithms that combine CR with other tridiagonal solvers such as Parallel Cyclic Reduction (PCR) or Recursive Doubling (RD). More recently, H. Kim et al. [80] have analyzed other hybrid algorithms and found that a combination of PCR and Thomas gave the best overall performance.

The rest of this chapter is structured as follows. Section 3.2 introduces the extended block cyclic reduction algorithm used by the BLKTRI solver. Section 3.4 gives a brief description of the standard algorithms for solving scalar tridiagonal systems. In Section 3.5 we detail the mapping of the BLKTRI solver on multicore and GPUs and analyze their performance and then in Section 3.6 we extend our discussion to 3D problems. Finally, Section 3.7 concludes summarizing the most

relevant contributions.

3.2. Three dimensional elliptic systems

In this section, we explain the strategy followed to solve a classical 3D Poisson equation:

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial z^2} = f(x,y,z)$$

defined on a Cartesian domain Ω with prescribed conditions on its boundary $\partial\Omega$.

Discretizing the domain using a Cartesian mesh uniform along each direction, for each (i,j,k) interior node we obtain:

$$\delta_x^2(i,j,k) + \delta_y^2(i,j,k) + \delta_z^2(i,j,k) = f_{i,j,k} \tag{3.1}$$

where

$$\begin{aligned}
\delta_x^2(i,j,k) &= \left(u_{i-1,j,k} - 2u_{i,j,k} + u_{i+1,j,k}\right)/\Delta x^2 \\
\delta_y^2(i,j,k) &= \left(u_{i,j-1,k} - 2u_{i,j,k} + u_{i,j+1,k}\right)/\Delta y^2 \\
\delta_z^2(i,j,k) &= \left(u_{i,j,k-1} - 2u_{i,j,k} + u_{i,j,k+1}\right)/\Delta z^2
\end{aligned}$$

are the finite difference centred approximations to the second derivatives along each direction. The boundary conditions that we will consider are either of Dirichlet or Neumann type on the surfaces normal to the y and z directions and periodic in the x one. The periodic condition aplied in one of the directions allows to

uncouple the 3D problem into a set of several independent 2D problems (Figure 3.1) using a discrete Fourier transform. Hereafter we will briefly explain how the decoupling process takes place. Let N being the number of equispaced nodes in the x direction that cover the interval $(0, 2\pi)$. We expand the uknown function $u(x,y,z)$ and $f(x,y,z)$ in Fourier series as:

$$u_{n,j,k} = \frac{1}{N}\sum_{l=1}^{N} \hat{u}_{l,j,k} e^{-i\alpha(n-1)} \text{ with } \alpha = \frac{2\pi(l-1)}{N} \tag{3.2}$$

where $\hat{u}_{l,j,k}$ is the l^{th} Fourier coefficient of the expansion. Next, the expansion is used in equation (3.1), obtaining the relationship:

$$\frac{1}{N}\sum_{l=1}^{N} e^{-i\alpha(n-1)} \left\{ \frac{\hat{u}_{l,j,k}}{\Delta x^2}(e^{-i\alpha} - 2 + e^{i\alpha}) + \delta_y^2 \hat{u}_{l,j,k} + \delta_z^2 \hat{u}_{l,j,k} \right\} = \frac{1}{N}\sum_{l=1}^{N} \hat{F}_{l,j,k} e^{-i\alpha n} \tag{3.3}$$

Equation (3.3) is equivalent to the set of N equations ($l = 1 \cdots N$):

$$\frac{\hat{u}_{l,j,k}(2cos(\alpha) - 2)}{\Delta x^2} + \frac{\hat{u}_{l,j+1,k} - 2\hat{u}_{l,j,k} + \hat{u}_{l,j-1,k}}{\Delta y^2} + \frac{\hat{u}_{l,j,k+1} - 2\hat{u}_{l,j,k} + \hat{u}_{l,j,k-1}}{\Delta z^2} = \hat{F}_{l,j,k} \tag{3.4}$$

having used the identity $e^{i\alpha} + e^{-i\alpha} = 2cos(\alpha)$. In short notation (3.4) reads as:

$$\frac{\hat{u}_{l,j+1,k} + \hat{u}_{l,j-1,k}}{\Delta y^2} + \frac{\hat{u}_{l,j,k+1} + u_{l,j,k-1}}{\Delta z^2} + \beta_l \hat{u}_{l,j,k} = \hat{F}_{l,j,k}, \;\; l = 1 \cdots N \tag{3.5}$$

with $\beta_l/2 = cos(\alpha) - 1/\Delta x^2 - 1/\Delta y^2 - 1\Delta z^2$. Thus, by considering the Fourier transform (direct FFT) of F one obtains a set of N, 2D independent problems having as unkwowns the Fourier coefficients $\hat{u}_{l,j,k}, l = 1..N$. Each independent problem concerns the solution of a linear system of equations which coefficient matrix is block tridiagonal. Of course, each one of this linear systems can now be solved in

a distributed fashion, in parallel. Once the solution is obtained in Fourier space a backward FFT can be used to recast the solution in physical space. Figure (3.1) provides an algorithmical sketch of the method.

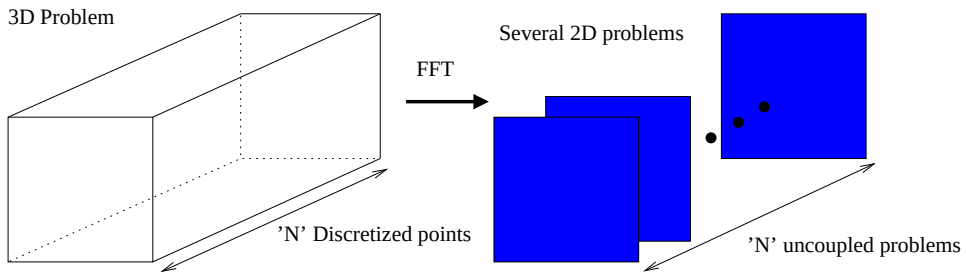


Figure 3.1: The Fourier based decoupling algorithm

To deal with each decoupled 2D problem, we have chosen a direct method based on a block cyclic reduction algorithm. As shown above, the whole method provides for a blend of coarse and fine-grain parallelism that can be exploited when mapped on heterogeneous platforms.

3.3. Extended block cyclic reduction

In this section we briefly summarize a classical direct method for the discrete solution of separable elliptic equations based on a block cyclic reduction algorithm [136]. This method is commonly used when tackling the solution of a linear system of equations arising from the second order centered finite difference discretization of 2D separable elliptic equations. From the standpoint of computational complexity (speed and storage), for a $m \times n$ net, its operation count is proportional to $mn \log_2 n$, and the storage requirements are minimal, since the solution is returned in the storage occupied by the right side of the equation (i.e., $m \times n$ locations are

required). More in details, consider the 2D separable elliptic equation having $u(x,y)$ as unknown field (Poisson equation is a particular case of what follows):

$$\frac{\partial}{\partial x}\left(a(x)\frac{\partial u}{\partial x}\right)+b(x)\frac{\partial u}{\partial x}+c(x)u+\frac{\partial}{\partial y}\left(d(y)\frac{\partial u}{\partial y}\right)+e(y)\frac{\partial u}{\partial y}+f(y)u=g(x,y) \tag{3.6}$$

If we discretize (3.6) with given Dirichlet or Neumann boundary conditions assigned on the edges of a square, using the usual five-point scheme with the discrete variables ordered in a lexicographic fashion, we obtain a linear system of $m \times n$ equations (having m nodes in the x direction and n in y one): $\mathbf{A}\tilde{\mathbf{u}} = \tilde{\mathbf{g}}$, where $\mathbf{A}$ is a block tridiagonal matrix:

$$\mathbf{A}=\begin{bmatrix} B_1 & C_1 & & & & 0 \\ A_2 & B_2 & C_2 & & & \\ & \cdot & \cdot & \cdot & & \\ & & \cdot & \cdot & \cdot & \\ & & & A_{n-1} & B_{n-1} & C_{n-1} \\ & & & & A_n & B_n \end{bmatrix}$$

and the vectors $\vec{u}$ and $\vec{g}$ are consistently split as a set of sub-vectors $\vec{u}_j$ and $\vec{g}_j$, $j = 1 \cdots n$, of length m each (i.e., the solution along the j^{th} domain row):

$$\mathbf{u} = [\tilde{\mathbf{u_1}},\ \tilde{\mathbf{u_2}}, \cdots\cdots \tilde{\mathbf{u_n}}]^{\mathbf{T}}$$

$$\mathbf{g} = [\tilde{\mathbf{y_1}},\ \tilde{\mathbf{y_2}}, \cdots\cdots \tilde{\mathbf{y_n}}]^{\mathbf{T}}$$

There is no restriction on m; however, cyclic reduction algorithms require $n =$

2^k, with large values of k for optimal performances. The blocks $\mathbf{A_i}$, $\mathbf{B_i}$ and $\mathbf{C_i}$ are $m \times m$ square matrices. In particular, the *BLKTRI* algorithm requires them to be of the form:

$$A_i = a_i I \tag{3.7}$$

$$B_i = B + b_i I \tag{3.8}$$

$$C_i = c_i I \tag{3.9}$$

where a_i, b_i and c_i are scalars. Having used a standard five point stencil for the discretization of (3.6), the matrix B is of tridiagonal pattern. The solution is obtained using an *extended cyclic reduction algorithm* which consists of the following phases (more details are found in [136]):

1. **Preprocessing**. This phase consists of computing the roots of certain matrix polynomials. This set of intermediate results only depends on the entries of **A** (not on the right hand side *rhs* of the equation).

2. **Recursive Reduction**. A sequence of linear systems is generated starting from the original complete one by decoupling odd and even equations. At each step, or level r, about half the unknown vector $\vec{u_i}$ is reduced by eliminating essentially half the remaining unknown vectors until a single unknown vector $\vec{u_2^k}$ remains.

3. **Back-substitution**. The solution vectors $\vec{u_i}$ are determined by first solving the final system generated in the above phase $\vec{u_2^k}$. Then the linear systems are

solved in reverse order determining more $\vec{u}_i$ solution vectors, using those $\vec{u}_i$ previously computed.

Overall, during the reduction phase the following equations are solved [136]:

$$q_i^{(r)} = (B_i^r)^{-1} B_{i-2^{r-1}}^{r-1} B_{i+2^{r-1}}^{r-1} p_i^{(r)} \tag{3.10}$$

$$p_i^{(r+1)} = \alpha_i^r (B_{i-2^{r-1}}^{r-1})^{-1} q_{i-2^r}^{(r)} + \gamma_i^r (B_{i+2^{r-1}}^{r-1})^{-1} q_{i+2^r}^{(r)} - p_i^r \tag{3.11}$$

where **g** is split in two different terms, q and p. B stores the roots calculated in the preprocessing phase. This procedure is required to stabilize the method [136]. α and γ have the following form:

$$\alpha_i^{(r)} = \prod_{j=i-2^{r+1}}^{i} aj \tag{3.12}$$

$$\gamma_i^{(r)} = \prod_{j=1}^{i+2^{r-1}} cj \tag{3.13}$$

Conversely, in the last phase the following equations are solved [136]:

for $r = k, k-1, \ldots, 0$ and $i = 2^r, 3\times 2^r, 5\times 2^r, \ldots, 2^{k-r}\times 2^r$:

$$u_i = (B_i^r)^{-1} B_{i-2^{r-1}}^{r-1} B_{i+2^{r-1}}^{r-1} \left[p_i^{(r)} - \alpha_i^r (B_{i-2^{r-1}}^{r-1})^{-1} u_{i-2^r} - \gamma_i^r (B_{i+2^{r-1}}^{r-1})^{-1} u_{i+2^r} \right] \tag{3.14}$$

This method is implemented in a FORTRAN package library called FISHPACK (the *BLKTRI* routine), which is widely used and well known within the computational fluid dynamics community [48].

To obtain the aforementioned terms the solution of a set of scalar tridiagonal systems of equations must be faced. The solution of these systems represent the most expensive stage of the algorithm. Nevertheless, other basic mathematical operations such as vectors sums or scalar vector multiplications introduce a non negligible cost.

3.4. Parallel tridiagonal algorithms

As mention above, a key element of the BLKTRI algorithm is how to solve a set of scalar tridiagonal systems. The original BLKTRI implementation in the *fishpack* package makes use of the Thomas algorithm (TA) [121]. TA is a specialized application of the Gaussian elimination that takes into account the tridiagonal structure of the system. TA consists of two stages, commonly denoted as forward elimination and backward substitution.

Given a linear $Au = y$ system, where A is a tridiagonal matrix:

$$A = \begin{bmatrix} b_1 & c_1 & & & & 0 \\ a_2 & b_2 & c_2 & & & \\ & \ddots & \ddots & \ddots & & \\ & & \ddots & \ddots & \ddots & \\ & & & a_{n-1} & b_{n-1} & c_{n-1} \\ & & & & a_n & b_n \end{bmatrix}$$

The forward stage eliminates the lower diagonal as follows:

$$c'_1=\frac{c_1}{b_1},\quad c'_i=\frac{c_i}{b_i-c'_{i-1}a_i}\quad \text{for } i=2,3,\ldots,n-1$$
$$y'_1=\frac{y_1}{b_1},\quad y'_i=\frac{y_i-y'_{i-1}a_i}{b_i-c'_{i-1}a_i}\quad \text{for } i=2,3,\ldots,n-1$$

and then the backward stage recursively solve each row in reverse order:

$$u_n=y'_n, u_i=y'_i-c'_iu_{i+1} \text{ for } i=n-1,n-2,\ldots,1$$

Overall, the complexity of TA is optimal: $8n$ operations in $2n-1$ steps. Unfortunately, this algorithm is purely sequential.

Cyclic Reduction (CR) [69, 166, 80] is a parallel alternative to TA. It also consists of two phases (reduction and substitution). In each intermediate step of the reduction phase, all even-indexed (i) equations $a_ix_{i-1}+b_ix_i+c_ix_{i+1}=d_i$ are reduced. The values of a_i, b_i, c_i and d_i are updated in each step according to:

$$a'_i=-a_{i-1}k_1, b'_i=b_i-c_{i-1}k_1-a_{i+1}k_2\ c'_i=-c_{i+1}k_2, y'_i=y_i-y_{i-1}k_1-y_{i+1}k_2$$

$$k_1=\frac{a_i}{b_{i-1}}, k_2=\frac{c_i}{b_{i+1}}$$

After $\log_2 n$ steps, the system is reduced to a single equation that is solved directly. All odd-indexed unknowns x_i are then solved in the substitution phase by introducing the already computed u_{i-1} and u_{i+1} values:

$$u_i=\frac{y'_i-a'_ix_{i-1}-c'_ix_{i+1}}{b'_i}$$

Overall, the CR algorithm needs $17n$ operations and $2\log_2 n-1$ steps. Figure 3.2 graphically illustrates its access pattern.

Parallel Cyclic Reduction (PCR) [70, 166, 80] is a variant of CR, which only has substitution phase. For convenience, we consider cases where $n=2^s$, that involve $s=\log_2 n$ steps. Similarly to CR a, b, c and y are updated as follows, for $j=1,2,\ldots,s$ and $k=2^{j-1}$:

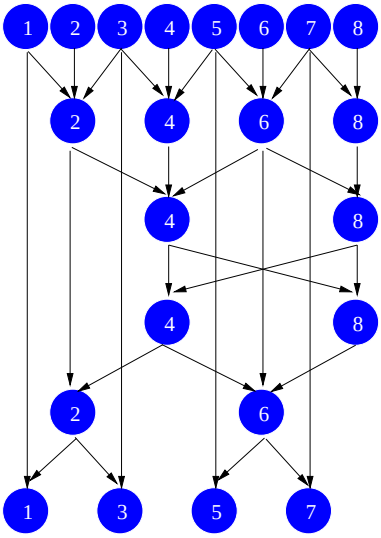


Figure 3.2: Access pattern of the CR algorithm.

$$a'_i = \alpha_i a_i, b'_i = b_i + \alpha_i c_{i-k} + \beta_i a_{i+k}$$
$$c'_i = \beta_i c_{i+1}, y'_i = b_i + \alpha_i y_{i-k} + \beta_i y_{i+k}$$

$$\alpha_i = \frac{-a_i}{b_{i-1}}, \beta_i = \frac{-c_i}{b_i}$$

finally the solution is achieved as:

$$u_i = \frac{y'_i}{b_i}$$

Essentially, at each reduction stage, the current system is transformed into two smaller systems and after $\log_2 n$ steps the original system is reduced to n independent equations. Overall, the operation count of PCR is $12n\log_2 n$. Figure 3.3 sketches the corresponding access pattern.

We should highlight that apart from their computational complexity these algorithms differ in their data access and synchronization patterns, which also have a strong influence on their actual performance. For instance, in the CR algorithm

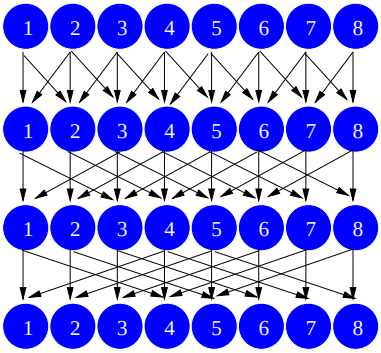


Figure 3.3: Access pattern of the PCR algorithm.

synchronizations are introduced at the end of each step and its corresponding memory access pattern may cause bank conflicts. PCR needs less steps and its memory access pattern is more regular [166]. In fact, hybrid combinations that try to exploit the best of each algorithm have been explored [166, 116, 34, 80]. Figure 3.4 illustrates the access pattern of the CR-PCR combination proposed in [166]. CR-PCR reduces the system to a certain size using the forward reduction phase of CR and then solves the reduced (intermediate) system with the PCR algorithm. Finally, it substitutes the solved unknowns back into the original system using the backward substitution phase of CR.

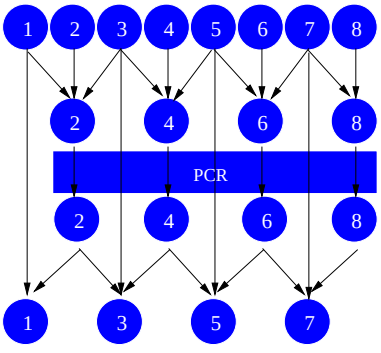


Figure 3.4: Communications pattern for the CR-PCR algorithm.

3.5. Parallel block cyclic reduction

The reduction and back-substitution phases are the core of the BLKTRI algorithm. In this section we focus on describing how to map these phases onto GPUs. The mapping on multicore systems requires less transformations to the original BLKTRI code. In this case, the most effective scheme consists in using a coarse-grain strategy for distributing the independent tridiagonal problems that arise at the different steps of the algorithm across the different cores. This way, these tridiagonal systems are solved sequentially on each core using the optimal TA algorithm. This distribution is well balanced and data locality is optimized mapping a subset of continuous systems onto each core. The original FISHPACK BLKTRI routine can be easily parallelized with this approach annotating some of its loops with Open-MP pragmas.

For our mapping on GPUs, we have identified four main kernels, which are graphically illustrated, along with their dependencies, in Figure 3.5. All data need to be uploaded to the GPU memory before launching the q kernel and finally, the solution u is transferred back to the CPU memory. For convenience, we have denoted $\alpha_i^r (B_{i-2^{r-1}}^{r-1})^{-1} q_{i-2^r}^{(r)}$ as α and $\gamma_i^r (B_{i+2^{r-1}}^{r-1})^{-1} q_{i+2^r}^{(r)}$ as γ. The p kernel consists on the addition of three vectors. The core of the computation is performed in the remaining three kernels, which share a similar pattern sketched in the *Generic Tridiagonal Kernel*. Essentially, these three kernels solve an independent set of tridiagonal systems but differ on their pre- and post-processing calculations.

Figure 3.6 illustrates with more detail the mapping of the generic kernel on the GPU. Figure 3.6-top shows a coarse-grain scheme similar to the multicore counterpart. In this coarse distribution a set of tridiagonal systems is mapped

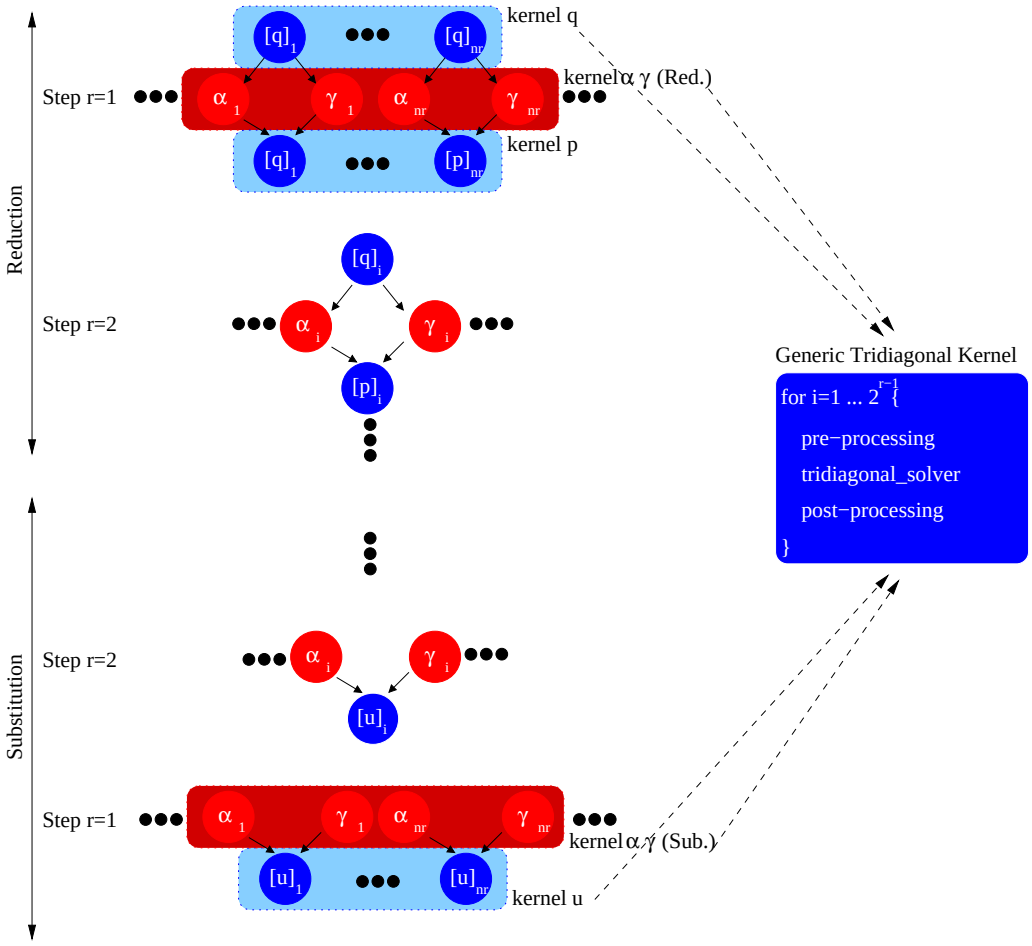


Figure 3.5: Main kernels of the reduction and substitution phases of the BLKTRI algorithm.

onto a CUDA block so that each CUDA thread fully solves a system using the TA algorithm. Unfortunately, this approach, which is relatively easy to implement, does not exploit efficiently the memory hierarchy of the GPU since the memory footprint of each CUDA thread becomes too large. Previous research has shown that fine-grain alternatives based on PCR are more efficient [166, 80, 152]. In this case (Figure 3.6-bottom), each tridiagonal system is distributed across the threads of a CUDA block so that the shared memory of the GPU can be used more effectively

(both the matrix coefficients and the right hand side of each tridiagonal system are hold on the shared memory of the GPU).

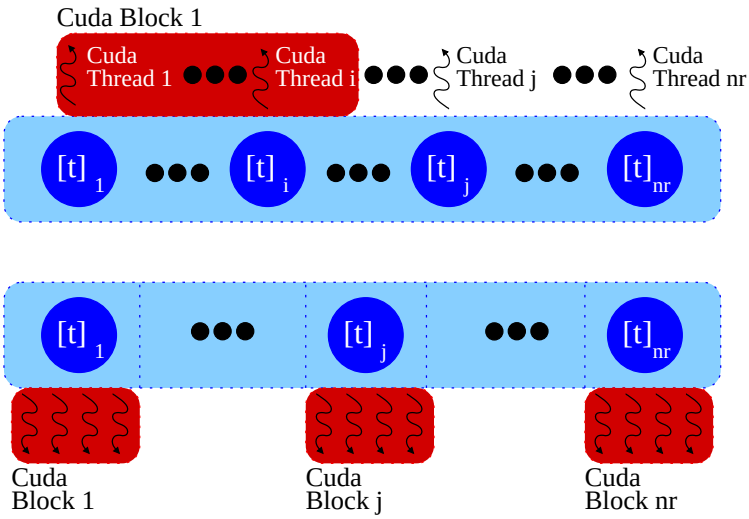


Figure 3.6: Coarse (top) and fine (bottom) distributions of the generic kernel.

We should highlight that the arithmetic intensity of our generic tridiagonal kernels is higher than the synthetic tridiagonal benchmarks analyzed by previous research [166, 80, 133]. This is an advantage when using the GPU as an accelerator since the impact of CPU-GPU data transfers on performance is much lower.

Figure 3.7-top (strong scaling) compares the different approaches using as a simplified test a single 1024×1024 2D problem. This is a relatively small 2D problem but note that it arises from the solution of a 3D problem. These tests have been run on an heterogeneous platform, whose main features are 2 CPUs Intel E5-2650 (up to 8 cores and 16 threads per processor) with 128 GB DDR3-1600 of RAM memory and 1 GPU nVidia K20c (Kepler) with 2496 CUDA cores and 5 GB GDDR5 of device memory. We have used Fedora Linux 16 and the compiler The Portland Group (PGI) Fortran (flags -fast -Mipa=fast,inline -mp -Mcuda). Each step of the algorithm has a different level of parallelism but, as shown in Figure 3.7-top,

the computational load of all of them are similar since as the level of parallelism reduces, the number of *iterations* of the generic tridiagonal kernel increases. The parallel implementations outperform the sequential BLKTRI routine in most cases. Only for the steps with reduced parallelism, the GPU version becomes ineffective. This is the expected behavior since in these cases the number of CUDA blocks is very small, being just one CUDA block in the last (first) step of the reduction (substitution) phase.

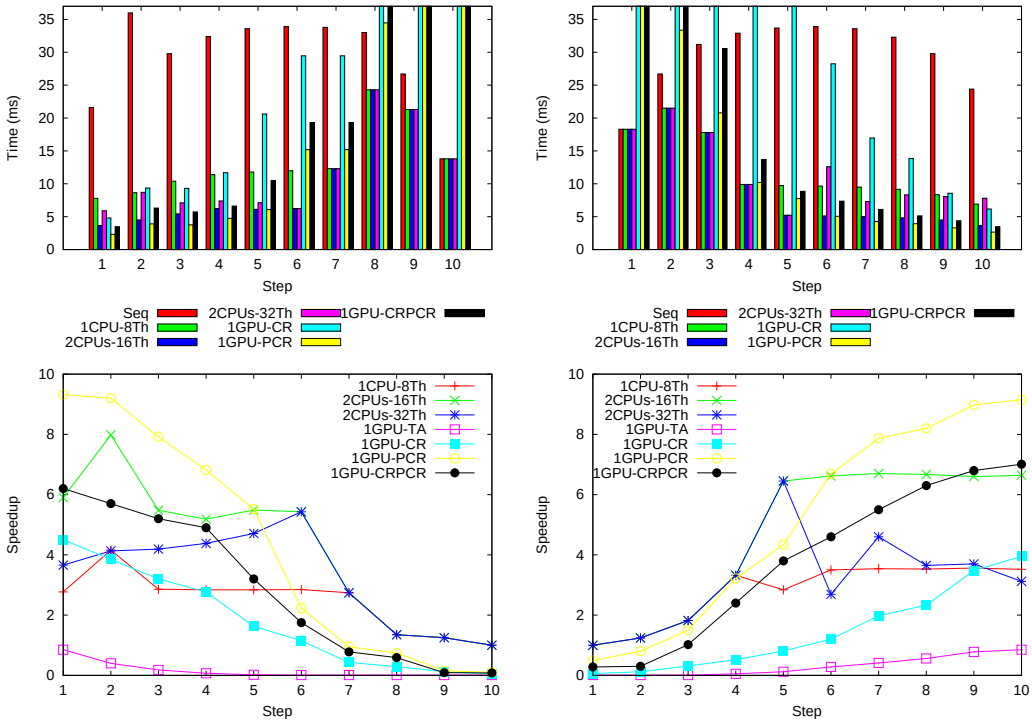


Figure 3.7: Execution time (top) and Speedup (bottom) on each step of the reduction (left) and substitution (right) phases of the extended 2D block cyclic reduction.

Figure 3.7-bottom shows the speedup at each step of the extended block cyclic reduction algorithm over the sequential counterpart. As mention above, the coarse thread distribution based on the TA algorithm does not perform well on GPUs, but on multicore, this coarse approach does provide satisfactory speedups across all

steps despite this is a small problem, achieving best performance when running 16 threads on 2 CPUs and 16 cores. On GPUs, PCR provides satisfactory speedups on the first (last) steps of the reduction (substitution) phases and is able to outperform CR and the hybrid CR-PCR algorithms across all steps.

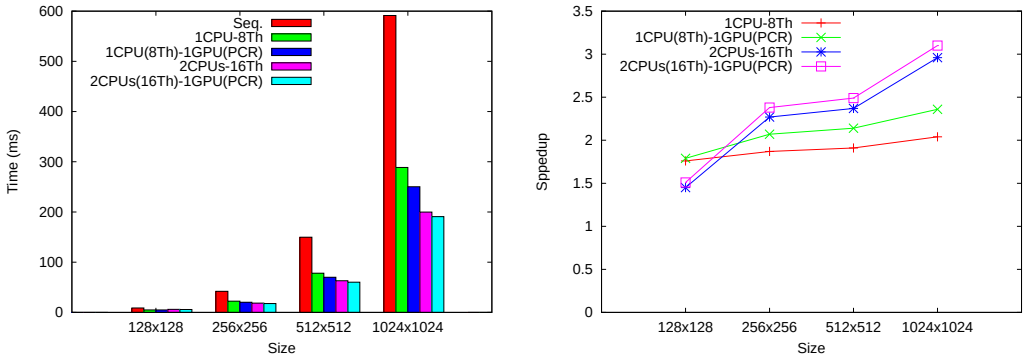


Figure 3.8: Total execution time (left) and trend of the speedup (right) increasing the size of the problem.

According to these results, the optimal approach appears to be an heterogeneous combination of PCR on the GPU and TA (16 threads) on multicore for those steps with lower parallelism. Nevertheless, this combination requires additional CPU-GPU data transfers that may degrade the actual performance. Figure 3.8 shows the overall speedups on Kepler for different problem sizes taking into account these data transfer overheads. We have fixed 5 different computing platforms and increased the size of the problem to carry out a weak scaling study. The heterogeneous approach is able to outperform the homogeneous multicore counterpart in all cases. A fully homogeneous GPU implementation (not shown in Figure 3.8) does not provide satisfactory results since in those steps with low parallelism, its performance becomes very inefficient.

3.6. Parallel three dimensional elliptic systems

In this Section we present the proposed approaches to solve in parallel a Three Dimensional Elliptic Systems problem on heterogeneous platforms. The FFT method can be computed in parallel on both multicore and GPUs platforms. This is a well know problem and there are several libraries that provide satisfactory results [47, 46, 30]. We have focused instead on solving the set of independent 2D problems in parallel, which is the main contribution of this work. In the 2D case, the homogeneous GPU implementation does not provide satisfactory results and the heterogeneous counterpart is able to achieve the best performance. We want to know if this is still valid for the 3D problem.

Figure 3.9-left graphically sketches the parallel profile of a 2D problem. We have highlighted three different stages: two of them have a high level of parallelism (blue areas) while the red one has limited parallelism as explained in the previous Section. In the 3D case (Figure 3.9-right) we need to solve a set of independent 2D problems and hence, the amount of parallelism increases in all the steps by a *problem size* factor, which we have denoted as the *S* factor.

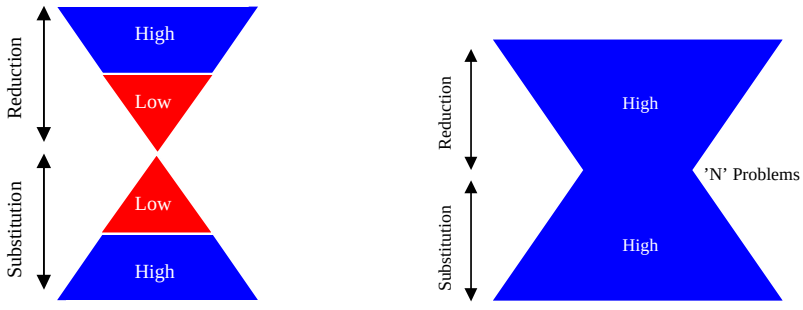


Figure 3.9: Amount of parallelism for one 2D block tridiagonal problem (left) and for a set of independent 2D block tridiagonal problems (right).

The mappings of a 3D problem on our target computing platforms are similar to

their 2D counterparts. On multicore, we follow a coarse-grain approach mapping a set of 2D problems on each core, which are solved sequentially using the TA algorithm. On GPUs, we follow a fine-grain approach based on the PCR algorithm, which is essentially the same as the 2D case. The major different is the number of CUDA blocks at each step, which is S times higher in 3D. An heterogeneous combination of the multicore and GPU implementation is also possible, as in the 2D case. The data transfers overheads are potentially much lower than in the 2D case since it is possible to perform them asynchronously. As shown graphically in Figure 3.10, this allows the overlapping of data transfers with useful computation on the GPUs or the CPUs, and indirectly, of GPUs with CPUs computation.

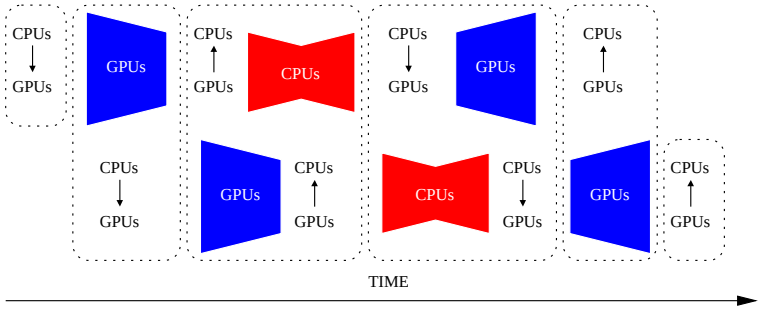


Figure 3.10: Heterogeneous approach.

Figure 3.11 (strong scaling) compares the homogeneous multicore and GPU implementations using as a test case a $512 \times 512 \times 512$ problem. Unlike the 2D case, the speedup figures are less dependent on the step of the algorithm due to the higher level of parallelism. In fact, in all the steps the speedup figures are close to the highest speedup attainable in the 2D case. Another important consequence is that the GPU version always outperforms the multicore counterpart.

These results question the potential benefits that the heterogeneous strategy could achieve since it seems that the homogeneous GPU implementation is able

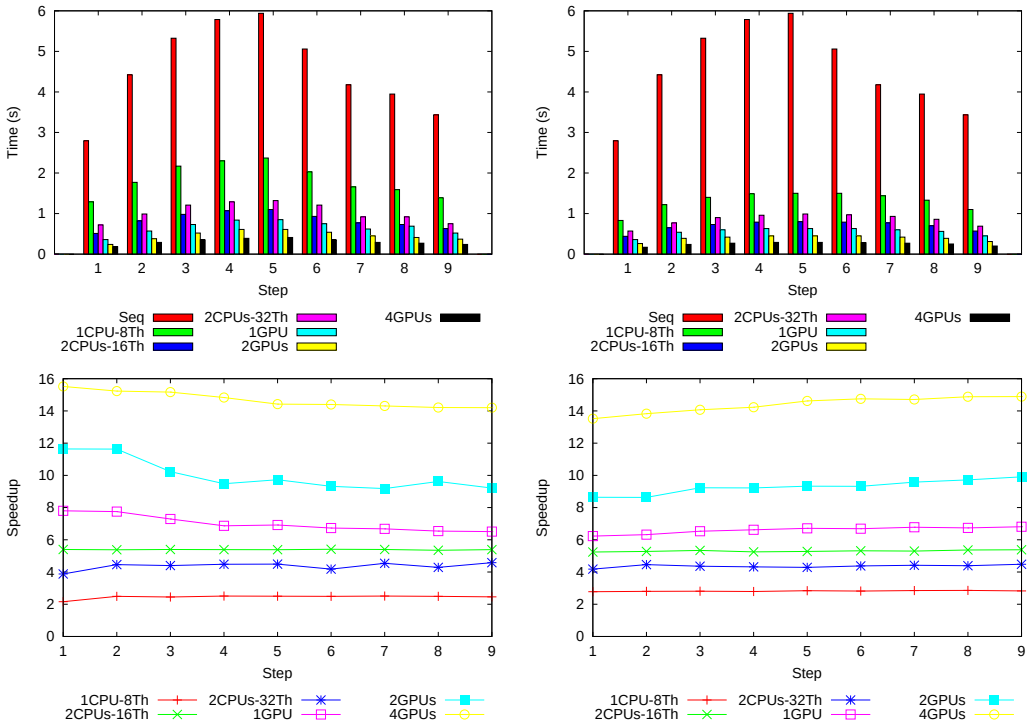


Figure 3.11: Execution time (top) and speedup (bottom) obtained in each step in the reduction (left) and substitution (right) phases.

to exploit all the available parallelism. Figure 5.19 (weak scaling) compares the homogeneous and heterogeneous approaches. First, we should highlight that in the homogeneous GPU implementation data transfers incur a very small overhead (lower than 2% of the execution time). This is the expected behavior since the arithmetic intensity of the 3D problem is very high. In spite of this, the heterogeneous approach is able to outperform the homogeneous counterpart since it benefits from actual GPUs-CPUs overlapping. As shown in Figure 5.19, these gains grow with the "S" factor. Overall, the extra benefit of the heterogeneous implementation can reach up to 15 % in terms of execution time over the homogeneous GPU counterpart.

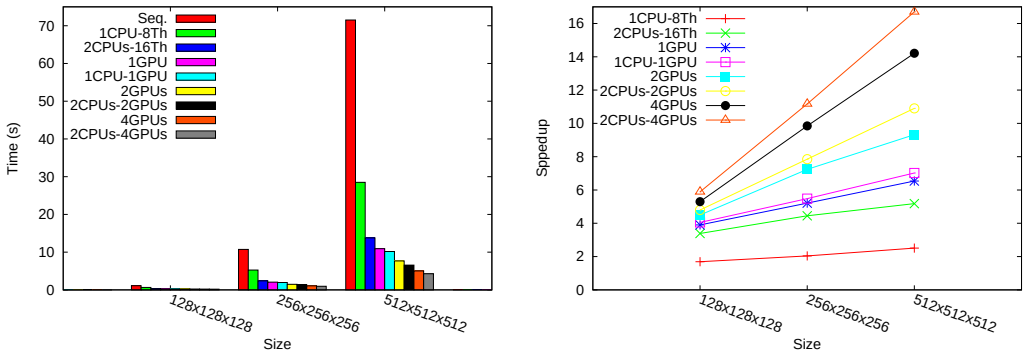


Figure 3.12: Execution time (s) (left) and speed up (right) for all the implementations.

3.7. Concluding remarks

The efficient solution of block tridiagonal linear systems is of crucial importance since it is the major bottleneck of several large scale simulation codes dealing with time-dependent elliptic partial differential equations. In this chapter we have analyzed the performance of different parallel implementation that exploit homogeneous multicores and GPUs systems, as well as heterogeneous multicores-GPUs platforms.

On multicore, a coarse grain approach based on the Thomas algorithm is the best option for solving the intermediate scalar tridiagonal systems that arise on both 2D and 3D problems. In contracts, on GPUs it is much better a fine grain alternative based on using the PCR algorithm.

As expected, a hybrid approach that combines both implementations is the best option on 2D problems. For 3D problems, we have shown that this is also the best choice despite the 3D problem has both higher arithmetic intensity and higher parallelism than the 2D case. Indeed, the homogeneous GPUs implementation outperform the multicore counterpart even for medium size 3D problems. However,

the hybrid approach benefits from CPUs-GPUs overlapping and it is able to achieve an additional 15% performance gain.

Unfortunately, the overall performance of those parallel fast solvers do not meet the goals envisioned in our research. This is why, in the next chapters we have tried to overcome the intrinsic limitation of Navier-Stokes solvers studying as an alternative the Lattice-Boltzmann method (LBM).

Chapter 4

Accelerating solid-fluid interaction using Lattice-Boltzmann and Immersed-Boundary coupled simulations on heterogeneous platforms

We propose a numerical approach based on the Lattice-Boltzmann (LBM) and Immersed Boundary (IB) methods to tackle the problem of the interaction of solids with an incompressible fluid flow. The proposed method uses a Cartesian uniform grid that incorporates both the fluid and the solid domain. This is a very optimum and novel method to solve this problem and is a growing research topic in Computational Fluid Dynamics. We explain in detail the parallelization of the whole method on both GPUs and an heterogeneous GPU-Multicore platform and

describe different optimizations, focusing on memory management and CPU-GPU communication. Our performance evaluation consists of a series of numerical experiments that simulate situations of industrial and research interest. Based on these tests, we have shown that the baseline LBM implementation achieves satisfactory results on GPUs. Unfortunately, when coupling LBM and IB methods on GPUs, the overheads of IB degrade the overall performance. As an alternative we have explored an heterogeneous implementation that is able to hide such overheads and allows us to exploit both Multicore and GPU resources in a cooperative way.

4.1. Introduction

The main objective of this work consists of minimizing the overhead caused by the simulation of solid-fluid interaction on CPU-GPU heterogeneous platforms. In particular, it is proposed an CPU-GPU heterogeneous scheduler which distributes either to GPU or CPU the different parts of the whole solver depending on their parallel features.

The dynamics of a solid in flow field is a research topic currently enjoying growing interest in many scientific communities. It is intrinsically interdisciplinary (structural mechanic, fluid mechanic, applied mathematics, etc) and covers a broad range of applications (aeronautics, civil engineering, biological flows, etc). The number of works in this field reflects the growing importance of the study of the dynamics in the solid-fluid interaction [165, 24, 112, 84]. The use of GPU architectures to compute the fluid field is widely used within Computational Fluid Dynamics community due to the significant performance results achieved [16, 110, 167]. In contrast, the solid-fluid interaction has only recently gained wider interest.

Classical fluid solvers based on the unsteady incompressible Navier Stokes equations may turn out to be inefficient or difficult to tune to achieve maximum performance on these new parallel platforms [154, 151]. A choice that better meets the GPUs characteristics is based on modeling the fluid flow through the Lattice Boltzmann method (LBM). Several recent works have shown that the combination of GPU-based platforms and methods based on the LBM algorithm can achieve impressive performances due to the intrinsic characteristics of the algorithm. Certainly, the computing stages of LBM are amenable to fine grain paralelization in an almost straightfoward way (see for example [16, 110] and references therein). Nevertheless, no much works has been done to extend the parallel efficiency of LBM to cases involving geometries bounded by complex, moving or deformable boundaries. A very recent work that covers a subject closely related with the present contribution is the one by [167] where a new efficient 2D implementation of LBM method for fluids flowing in geometries with curved boundary using GPUs platforms is proposed. Curved boundaries are taken into account via a non equilibrium extrapolation scheme developed by [59]. Here, we will focus on a different approach based on LBM coupled with an Immersed Boundary method technique able to deal with complex, moving or deformable boundaries [102, 162, 71, 168, 169, 148, 3]. Special emphasis are given to the algorithmic and implementation techniques adopted to keep the solver highly efficient on CPU-GPU heterogeneous platforms.

This chapter is structured as follows. Section 5.2 briefly introduces the physical problem at hand and the general numerical framework that has been selected to cope with it: Lattice-Boltzmann method (LBM) coupled with Immersed-Boundary (IB) technique based on the use of a set of *Lagrangian* nodes distributes along

the solid boundaries. In Section 5.4 the specific potential parallel features of IB method are presented. In Section 5.5, we detail the parallel strategies envisaged to optimally enhance the performance of the global LBM-IB algorithm on CPU-GPU heterogeneous platforms. Finally, Section 5.6 details the performance analysis of the proposed techniques and in Section 5.7 some conclusions are outlined.

4.2. Mathematical formulation: Lattice-Boltzmann and Immersed-Boundary method

The Lattice Boltzmann method combined with an Immersed Boundary technique is highly attractive when dealing with moving or deformable bodies for two main reasons: the shape of the boundary, tracked by a set of Lagrangian nodes is a sufficient information to impose the boundary values; and the force of the fluid on the immersed boundary is readily available and thus easily incorporated in the set of equations that govern the dynamics of the immersed object. In addition, it is also particularly well suited for massively parallelized simulations, as the time advancement is explicit and the computational stencil is formed by few local neighbors of each computational node (support). The fluid is discretized on the regular Cartesian mesh while the shape of the solids is discretized in a Lagrange fashion by a set of points which obviously do not necessarily coincide with mesh points. The Lattice Boltzmann method has been extensively used in the past decades (see [135] for a complete overview) and now is regarded as a powerful and efficient alternative to classical Navier Stokes solvers. In what follows we briefly recall the basic formulation of the method. The LBM is based on an equation that governs

the evolution of a discrete distribution function $f_i(\mathbf{x},t)$ describing the probability of finding a particle at Lattice site $\mathbf{x}$ at time t with velocity $\mathbf{v} = \mathbf{e}_i$. In this work, we consider the *BGK* formulation that relies upon an unique relaxation time τ toward the equilibrium distribution $f_i^{(eq)}$:

$$f_i(\mathbf{x}+\mathbf{e}_i\Delta t, t+\Delta t) - f_i(\mathbf{x},t) = -\frac{\Delta t}{\tau}\left(f(\mathbf{x},t) - f_i^{(eq)}(\mathbf{x},t)\right) + \Delta t F_i \tag{4.1}$$

The particles can move only along the links of a regular Lattice defined by the discrete speeds ($e_0 = c(0,0)$; $e_i = c(\pm 1,0), c(0,\pm 1)$, $i = 1\cdots 4$; $e_i = c(\pm 1,\pm 1)$, $c(\pm 1, \pm 1)$, $i = 5\cdots 8$ with $c = \Delta x/\Delta t$) so that the synchronous particle displacements $\Delta\mathbf{x}_i = \mathbf{e}_i\Delta t$ never take the fluid particles away from the Lattice. For the present study, the standard two-dimensional 9-speed Lattice *D2Q9* is used , but all the techniques that will be presented can be extended in a straightforward manner to three dimensional lattices. The equilibrium function $f^{(eq)}(\mathbf{x},t)$ can be obtained by Taylor series expansion of the Maxwell-Boltzmann equilibrium distribution:

$$f_i^{(eq)} = \rho\omega_i\left[1 + \frac{\mathbf{e_i}\cdot\mathbf{u}}{c_s^2} + \frac{(\mathbf{e_i}\cdot\mathbf{u})^2}{2c_s^4} - \frac{\mathbf{u}^2}{2c_s^2}\right] \tag{4.2}$$

In equation 5.2, c_s is the speed of sound ($c_s = 1/\sqrt{3}$), ρ is the macroscopic density, and the weight coefficients ω_i are $\omega_0 = 4/9$, $\omega_i = 1/9$, $i = 1\cdots 4$ and $\omega_5 = 1/36$, $i = 5\cdots 8$ according to the current normalization. The macroscopic velocity $\mathbf{u}$ in equation 5.2 must satisfy a Mach number requirement $|\,\mathbf{u}\,|/c_s \approx M << 1$. This stands as the equivalent of the CFL number for classical Navier Stokes solvers. Finally, in 5.1, F_i represents the contribution of external volume forces at lattice level that in our case include the effect of the immersed boundary. Given any external volume force

$\mathbf{f}^{(ib)}(\mathbf{x},t)$, the contribution on the lattice are computed according to the formulation proposed by [59] as:

$$F_i = \left(1 - \frac{1}{2\tau}\right) \omega_i \left[\frac{\mathbf{e_i} - \mathbf{u}}{c_s^2} + \frac{\mathbf{e_i} \cdot \mathbf{u}}{c_s^4}\mathbf{e_i}\right] \cdot \mathbf{f_{ib}} \tag{4.3}$$

The multi-scale Chapman Enskog expansion of equation 5.1, neglecting terms of $O(\varepsilon M^2)$ and using expression 5.3, returns the Navier-Stokes equations with body forces and the kinematic viscosity related to lattice scaling as $\nu = c_s^2(\tau - 1/2)\Delta t$.

Without the contribution of the external volume forces stemming from the immersed boundary treatment, equation 5.1 is typically advanced in time in two stages, the collision and the streaming ones.

Given $f_i(\mathbf{x},t)$ compute:

$\rho = \sum f_i(\mathbf{x},t)$ and $\rho\mathbf{u} = \sum \mathbf{e}_i f_i(\mathbf{x},t)$

Collision stage:

$f_i^*(\mathbf{x},t+\Delta t) = f_i(\mathbf{x},t) - \frac{\Delta t}{\tau}\left(f(\mathbf{x},t) - f_i^{(eq)}(\mathbf{x},t)\right)$

Streaming stage:

$f_i(\mathbf{x}+\mathbf{e}_i\Delta t, t+\Delta t) = f_i^*(\mathbf{x},t+\Delta t)$

Depending on the ordering of the two stages two different strategies arise. The classical approach is known as the push method and performs collision before streaming. We have adopted instead for pull method [158] which performs the steps in the opposite order. This can lead to an important performance enhancement on fine grained parallel machines. A short discussion about the different implementations and achieved performances using the two orderings will be detailed later on.

We close this section by briefly explaining the Immersed Boundary method that we use both to enforce boundary values and to recover the fluid force exerted on immersed objects within the framework of the LBM algorithm [148, 3]. In the present IB approach as in several others, the fluid is discretized on a regular Cartesian lattice while the immersed objects are discretized and tracked in a Lagrangian fashion by a set of markers distributed along their boundaries. The general set up of the present Lattice Boltzmann–Immersed Boundary method can be recast in the following algorithmic sketch.

Given $f_i(\mathbf{x},t)$ compute:

$\rho = \sum f_i(\mathbf{x},t)$ and

$\rho\mathbf{u} = \sum \mathbf{e}_i f_i(\mathbf{x},t) + \frac{\Delta t}{2}\mathbf{f}_{ib}$

Collision stage:

$\hat{f}_i^* (\mathbf{x},t+\Delta t) = f_i (\mathbf{x},t) - \frac{\Delta t}{\tau}\left(f(\mathbf{x},t) - f_i^{(eq)}(\mathbf{x},t)\right)$

Streaming stage:

$\hat{f}_i (\mathbf{x}+\mathbf{e}_i\Delta t, t+\Delta t) = \hat{f}_i^* (\mathbf{x},t+\Delta t)$

Compute :

$\hat{\rho} = \sum \hat{f}_i(\mathbf{x}+\mathbf{e}_i\Delta t, t+\Delta t)$ and

$\hat{\rho}\hat{\mathbf{u}} = \sum \mathbf{e}_i \hat{f}_i(\mathbf{x}+\mathbf{e}_i\Delta t, t+\Delta t)$

Interpolate on Lagrangian markers (volume force):

$\hat{\mathbf{U}}(\mathbf{X}_k, t+\Delta t) = \mathscr{I}(\hat{\mathbf{u}})$ and

$\mathbf{f}_{ib}(\mathbf{x},t) = \frac{1}{\Delta t}\mathscr{S}\left(\mathbf{U}_d(\mathbf{X}_k, t+\Delta t) - \hat{\mathbf{U}}(\mathbf{X}_k, t+\Delta t)\right)$

Repeat collision with body forces (see 5.3) and Streaming:

$$f_i^*(\mathbf{x}, t+\Delta t) = f_i(\mathbf{x}, t) - \frac{\Delta t}{\tau}\left(f(\mathbf{x}, t) - f_i^{(eq)}(\mathbf{x}, t)\right) + \Delta t F_i \text{ and}$$

$$f_i(\mathbf{x} + \mathbf{e}_i \Delta t, t+\Delta t) = f_i^*(\mathbf{x}, t+\Delta t)$$

As outlined above, the basic idea consists in performing each time step twice. The first one, performed without body forces, allows to predict the velocity values at the immersed boundary markers and the force distribution that restores the desired velocity boundary values at their locations. The second one applies the regularized set of singular forces and repeats the procedure advancing (using 5.3) to determine the final values of the distribution function at the next time step. The key aspects of the algorithm and of its efficient implementation depend on the way the interpolation $\mathscr{I}$ and the $\mathscr{S}$ operators (termed as spread from now on) are applied. Here, following [148] and [3] we perform both operations (interpolation and spread) through a convolution with a compact support mollifier meant to mimic the action of a Dirac's delta. Combining the two operators we can write in a compact form:

$$\mathbf{f}_{(ib)}(\mathbf{x}, t) = \frac{1}{\Delta t}\int_\Gamma \left(\mathbf{U_d}(\mathbf{s}, t+\Delta t) - \int_\Omega \hat{\mathbf{u}}(\mathbf{y})\tilde{\delta}(\mathbf{y}-\mathbf{s})d\mathbf{y}\right)\tilde{\delta}(\mathbf{x}-\mathbf{s})d\mathbf{s} \tag{4.4}$$

where $\tilde{\delta}$ is the mollifier, to be defined later, Γ is the immersed boundary, Ω is the computational domain, and $\mathbf{U_d}$ is the desired value on the boundary at the next time step. The discrete equivalent of 5.4 is simply obtained by any standard composite quadrature rule applied on the union of the supports associated to each Lagrangian marker. As an example, the quadrature needed to obtain the force distribution on the lattice nodes is given by:

$$f_{ib}^l(x_i, y_j) = \sum_{n=1}^{N_e} F_{ib}^l(\mathbf{X_n})\tilde{\delta}(x_i - X_n, y_j - Y_n)\varepsilon_n \tag{4.5}$$

where the superscript l refers to the l^{th} component of the immersed boundary force, (x_i, y_j) are the lattice nodes (*Cartesian* points) falling within the union of all the supports, N_e is the number of Lagrangian markers and ε_n is a value to be determined to enforce consistency between interpolation and the convolution 5.5. More details about the method and in particular about the determination of the ε_n values can be found in [3]. In what follows we will give more details on the construction of the support cages surrounding each Lagrangian marker since it plays a key role in the parallel implementation of the IB algorithm. As already mentioned, the embedded boundary curve is discretized into a number of markers $\mathbf{X}_I,\ I = 1..N_e$. Around each marker $\mathbf{X}_I$ we define a rectangular cage Ω_I with the following properties: (i) it must contain at least three nodes of the underlying Eulerian lattice for each direction; (ii) the number of nodes of the lattice contained in the cage must be minimized. The modified kernel, obtained as a cartesian product of the one dimensional function [2]

$$\tilde{\delta}(r) = \begin{cases} \frac{1}{6}\left(5 - 3|r| - \sqrt{-3(1-|r|)^2+1}\right) & \\ & 0.5 \leq |r| \leq 1.5 \\ \frac{1}{3}\left(1 + \sqrt{-3r^2+1}\right) & 0.5|r| \leq 0.5 \\ 0 & \text{otherwise} \end{cases} \tag{4.6}$$

will be identically zero outside the square Ω_I. We take the edges of the square to measure slighlty more than three lattice spacings Δ (i.e., the edge size is $3\Delta + \eta = 3 + \eta$ in the actual LBM normalization). With such choice, at least three nodes of the lattice in each direction fall within the cage. Moreover a value of $\eta << 1$ ensures that the mollifier evaluated at all the nine (in two dimensions) lattice nodes takes on a non zero value. The interpolation stage is performed locally on each nine points

support: the values of velocity at the nodes withing the support cage centered about each Lagrangian marker deliver approximate values (i.e., second order) of velocity at the marker location. The force spreading step requires information from all the markers, typically spaced $\Delta = 1$ apart along the immersed boundary. The collected values are then distributed on the union of the supports meaning that each support may receive information from supports centered about neighboring markers, as in 5.5. The outlined method has been validated for several test cases including moving rigid immersed objects and flexible ones [41].

4.3. Immersed-Boundary on multicore and GPU platforms

This section presents the strategy that we have adopted for the efficient parallelization of the IB algorithm when executed on CPU-GPU heterogeneous platforms. The computations related with the *Lagrange* markers (support) distributed on the solid/s surface can be parallelized efficiently on both, CPU and GPU. As already mentioned the whole algorithm can be seen as a two steps procedure: a first, global LBM update, and a subsequent local correction to impose the boundary values. It is well known that the memory management plays a crucial role in the performance of parallel computing. To compute the IB method and the Body Force Introduction (BFI), it is necessary to store the information about the coordinates, velocities and forces of all the *Lagrangian* points and their supports. A set of memory management optimizations, which depends on the access pattern, have been carried out for the IB method implementation on both platforms, multicore

and GPU, to achieve an effective memory usage. In order to facilitate memory bandwidth exploitation and the parallel distribution of the workload, memory structures based on the style of C programming language have been used. Two different memory management approaches are proposed depending on the use of multicore or GPU, since both architecture show different memory features and hierarchy.

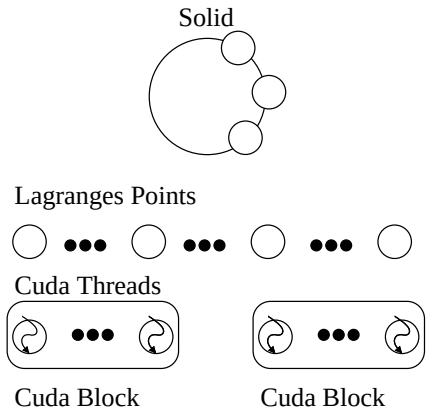


Figure 4.1: CUDA block-thread distribution for *Lagrangian* points.

The multicore approach stores the information of a particular *Lagrangian* point and its support in nearby memory locations, which benefits the exploitation of coarse grain parallelisms. In contrast, in order to achieve a coalescing access to global memory, the GPU approach distributes the information of all *Lagrangian* points in a set of one-dimensional arrays. In this way, continuous threads access to continuous memory locations.

Next, several approaches to implement the IB method are proposed. The degree of parallelism of the IB method is given by the number of *Lagrangian* points. The multicore approach carries out a coarse-grain parallelism by mapping a set of continuous *Lagrangian* points on each core which are solved sequentially. This distribution is well balanced and the use of the memory is optimized by using

the memory structures previously described.The set of *Lagrangian* points can be easily parallelized with this approach, annotating some of its loops with OpenMP pragmas.

On the GPU, the implementation consists of using 2 basic kernels. The first one, denoted as Immersed Forces Computation (IFC) kernel, assembles the velocity field on the supports, undertakes the interpolation at the *Lagrangian* markers and determine the *Eulerian* volume force field on each node of the union of the supports. The second kernel, denoted as Body Forces Computation (BFC) kernel, computes the lattice forces and repeat the LBM time update only on the union of the supports including the IB forces contribution.

Both kernels use the same CUDA block-thread distribution (Figure 5.6). The first kernel computes the whole IB method. It consists of computing these major steps:

1. Velocities interpolation. The input parameters of this step are loaded from global memory to local registers using coalesced memory accesses.

2. Force Computation. The parameters are held in both, local and global memory (coalesced accesses). The computed forces are held in local registers.

3. Spread the forces. The parameters are used from local and global memory and the results are stored in global memory by using atomic operations.

After the spreading step the forces are stored in the global memory by using *atomic* functions. These *atomic* functions are performed to prevent race conditions. Particularly, we used these operations to avoid incoherent executions, since the supports of different *Lagrangian* points can share the same *Eulerian* points.The pseudo-code of the IFC kernel is graphically illustrated in Algorithm 7.

Algorithm 3 IFC kernel.

```
1: IFC kernel(solid s,U_x,U_y)
2: vel_x, vel_y, force_x, force_y
3: for i = 1 → numSupport do
4:     vel_x += interpol(U_x[s.Xsupp[i]], s)
5:     vel_y += interpol(U_y[s.Ysupp[i]], s)
6: end for
7: force_x = computeForce(vel_x, s)
8: force_y = computeForce(vel_y, s)
9: for i = 1 → numSupport do
10:    AddAtom(s.XForceSupp, spread(force_x, s))
11:    AddAtom(s.YForceSupp, spread(force_y, s))
12: end for
```

After the execution of the IB related computations (IFC kernel), the lattice forces as in Equation 5.3 (Section 5.2) need to be determined. Before tackling this next stage it is necessary to introduce a synchronization point that guarantees that all the IB forces have been actually computed on all the points within the union of the supports. Nonetheless, the global memory access required to determine the system of lattice forces is larger than in the previous stage: 9 directions for each lattice node in the support. Also in this case to inhibit *race* conditions it has been necessary to resort to *atomic* functions. As for the case of the equilibrium distribution, also here the computation of the lattice force contributions is carried out using registers. The pseudo-code for this final kernel is given in algorithm 8.

4.4. Lattice-Boltzmann & Immersed-Boundary on CPU-GPU heterogeneous platforms

The actual computational scheduling of LBM is based on the work by [110], a novel efficient CUDA implementation based on a *pull* single-loop strategy: each

Algorithm 4 BFC kernel.

```
1: BFC kernel(solid s,f_x,f_y)
2: F_body(Body Force),g (Gravity),x,y,vel_x,vel_y
3: for i = 1 → numSupport do
4:     x = s.Xsupport[i]
5:     y = s.Ysupport[i]
6:     vel_x = s.VelXsupport[i]
7:     vel_y = s.VelYsupport[i]
8:     for j = 1 → 9 do
9:         F_body = (1 − 0.5 · 1/τ) · w[j] · (3 · ((c_x[j] − vel_x) · (f_x[x][y] + g) + (c_y[j] −
   vel_y) · f_y[x][y]))
10:        AddAtom(f^{n+1}[j][x][y], F_body)
11:    end for
12: end for
```

CUDA thread is uniquely dedicated to a single lattice node, performing one complete time step of LBM. In general, the *pull* method reorganizes the memory pattern access by changing the ordering of the LBM steps:

1. Move distribution functions $f_i(x+c_i\Delta t, t+\Delta t)$ values from global memory to local memory (coalescing accesses) and perform streaming.

2. Compute the macroscopic averages ρ, u (local memory).

3. Calculate the collision step $f_i^{(eq)}$ (local memory).

4. Copy the new values f_i into the global memory (coalescing accesses).

Our first parallel implementation of the LBM-IB method performs all the major steps on the GPU. The host CPU is used exclusively for a pre-processing stage that sets up the initial configuration and uploads those initial data to the GPU memory and a monitoring stage that downloads the information of each lattice node (i.e., velocity components and density) back to the CPU memory when required.

As shown in Figure 4.2-top, this implementation consists of three CUDA kernels denoted as LBM, IFC and BFC respectively, that are launched consecutively for every time step. The first kernel implements the LBM method while the other two perform the IB correction. The overhead of the preprocessing stage performed on the CPU is negligible and the data transfer of the monitoring stage are mostly overlapped with the execution of the LBM kernel.

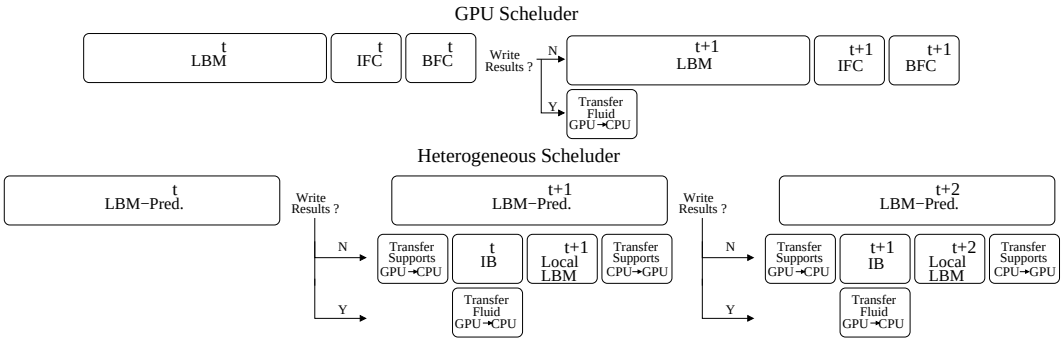


Figure 4.2: GPU (top) and CPU-GPU Heterogeneous (bottom) implementations.

Although this approach achieves satisfactory results, its speedups are substantially lower than those achieved by pure LBM solvers [110]. The obvious reason behind this behavior is the ratio between the characteristic volume fraction and the fluid field, which is very small. Therefore, the amount of data parallelism in the LBM kernel is substantially higher than in the other two kernels. In fact, for the target problems investigated, millions of threads compute the LBM kernel, while the IFC and BFC kernels only need thousands of them. But in addition, those kernels also require *atomic* functions due to the intrinsic characteristics of the IB method and those operations degrade performance.

As an alternative to mitigate those problems, we have explored a heterogeneous implementation graphically illustrated in Figure4.2-bottom. The LBM kernel is

computed on the GPU as in the previous approach but the whole IB method and an additional local correction to LBM on the supports of the *Lagrangian* points is performed on the CPU in a coordinated way using a pipeline. This way, we are able to overlap the prediction of the fluid field for the "$t+1$" iteration with the correction of the IB method on the previous iteration "t' at the expense of a local LBM computation of the"$t+1$" iteration on the CPU and additional transfers of the *supports* between the GPU and the CPU at every simulation step.

4.5. Performance evaluation

To critically evaluate the performance of the developed LBM and IB solver, next we consider a number of tests executed on a CPU-GPU (i.e., Xeon-Kepler) system. More details about the specific architectures that have been used for performance evaluation are given in Table 4.1. According to the memory requirements of the kernels, the memory hierarchy has been configured as 16KB shared memory and 48KB L1, since our codes do not benefit from a higher amount of shared memory on the investigated tests. All the simulations have been performed using double precision and as a performance metric we have used the conventional MFLUPS metric (millions of fluid lattice updates per second) used in most LBM studies.

Platform	Xeon E5520 (2.26 GHz)	Kepler K20c
Cores	8	2496
on-chip Memory	L1 32KB (per core)	SM 16/48KB (per MP)
	L2 512KB (unified)	L1 48/16KB (per MP)
	L3 20MB (unified)	L2 768KB (unified)
Memory	64GB DDR3	5GB GDDR5
Bandwidth	51.2 GB/s	208 GB/s
Compiler	gcc 4.6.2	nvcc 5.5

Table 4.1: Details of the experimental platforms.

The first tests (Figure 5.17) focus exclusively on the IB method using a synthetic simulation without considering the LBM method and analyze its acceleration on both multicore and GPUs. Even for a moderate number of *Lagrangian* nodes, we achieve substantial speedups over the sequential implementation on both platforms. Despite the overheads mentioned above, our GPU implementation is able to outperform the multicore counterpart (8 cores) from 2500 *Lagrangian* markers.

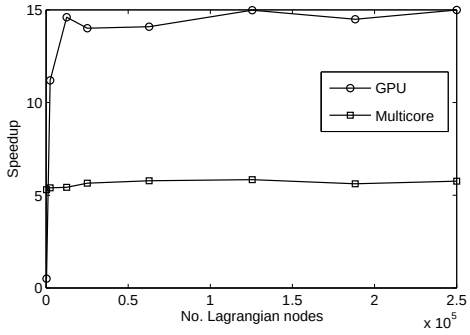


Figure 4.3: Speedups of the IB method on multicore and GPU for increasing number of Lagrangian nodes.

The performance of the whole LBM-IB solver is analyzed in Figure 5.6.4.1. We have used the same physical setting as in Section 5.2 with an increasing number of lattice nodes to analyzed the scalability of the method. We have investigated two realistic scenarios with characteristics volume fractions of 0.5% and 1% respectively (i.e. the amount of embedded *Lagrangian* markers also grows with the number of lattice nodes).

The performance of the homogeneous GPU implementation of the LBM-IB method drops substantially over the pure LBM implementation. The slowdown is around 15% for a solid volume fraction of 0.5%, growing to 25% for the 1% case. In contrast, for these fractions our heterogeneous approach is able to hide

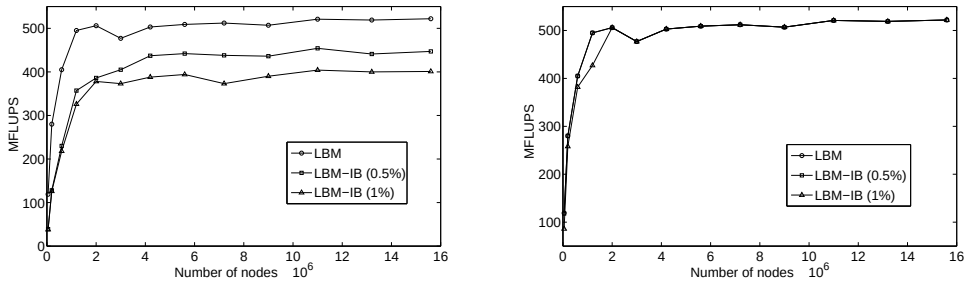


Figure 4.4: Performance of our GPU (left) and CPU-GPU (right) solvers in MFLUPS for the investigated simulations.

the overheads of the IB method, reaching similar performance to the pure LBM implementation.

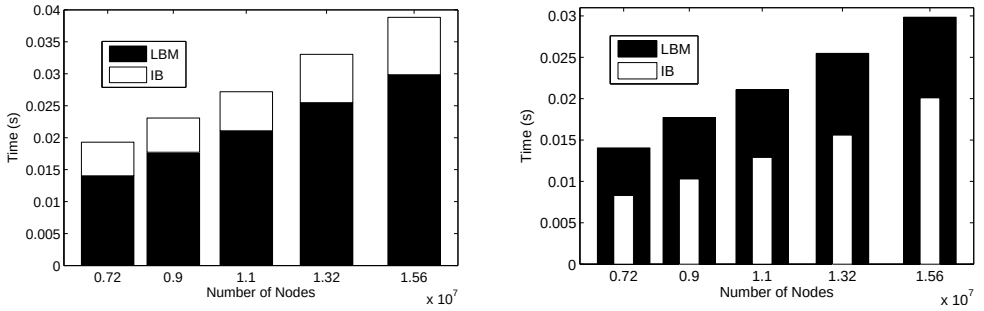


Figure 4.5: Execution time consumed by the LBM and IB method on both, the GPU homogeneous (left) and multicore-GPU heterogeneous (right) platforms.

Finally, Figure 5.19 illustrates the overhead of the IFC and BFC kernel in the GPU homogeneous approach (*left*) and the execution time consumed by the IB method and the local LBM corrections (*right*) over the pure LBM implementation in the heterogeneous approach for a solid volume fraction of 1%. As shown, the consumed time by the steps computed on multicore (i.e. IB method and local LBM corrections) in the multicore-GPU heterogeneous approach does not suppose an additional cost over the pure LBM solver, representing around 65% of the total time

consumed by the LBM kernel.

4.6. Concluding remarks

In this chapter we focus on a hybrid framework of a coupled Lattice-Boltzmann and Immersed Boundary method to simulate Fluid-Solid interaction. Our motivation consists of the design and development of a heterogeneous approach to mitigate the overhead introduced by the computing of the solid/s presence. Our approach is able to minimize the overhead of such simulations and match the performance of state-of-the-art pure LBM solvers. We have followed a heterogeneous approach, which distributes the execution of solid presence (IB) on multicore, while the LBM is computed on GPU at very same time. Given the good performance achieved, in the next chapter, we consider the use of other hardware accelerators as Intel Xeon Phi to deal with the same scenario. Despite Intel Xeon Phi is a hardware accelerator, as NVIDIA's GPUs, the differences found at hardware level, force us to propose new techniques to efficiently exploit the particular features of the hardware.

In this chapter we have investigated the performance of a coupled Lattice-Boltzmann and Immersed Boundary method that simulates the contribution of solid behavior within an incompressible fluid. While, the Lattice-Boltzmann method has been widely studied on heterogeneous platforms, the Immersed-Boundary method has received less attention.

Our main contribution has been the design and analysis of a heterogeneous implementation that takes advantage of both GPUs and multicore in a cooperative way. For realistic physical scenarios with realistic solid volume fractions, our heterogeneous solver is able to hide the overheads introduced by the Immersed-

Boundary method and match the performance (MFLUPS) of state-of-the-art pure LBM solvers.

In the next chapter we generalize this work with a more elaborated performance analysis and using other accelerators such as the Intel Xeon Phi.

Chapter 5

Accelerating fluid-solid simulations (Lattice-Boltzmann & Immersed-Boundary) on heterogeneous architectures

We propose a numerical approach based on the Lattice-Boltzmann (LBM) and Immersed Boundary (IB) methods to tackle the problem of the interaction of solids with an incompressible fluid flow, and its implementation on heterogeneous platforms based on data-parallel accelerators such as NVIDIA GPUs and the Intel Xeon Phi. We explain in detail the parallelization of these methods and describe a number of optimizations, mainly focusing on improving memory management and reducing the cost of host-accelerator communication. As previous research has consistently shown, pure LBM simulations are able to achieve good performance results on heterogeneous systems thanks to the high parallel efficiency of

this method. Unfortunately, when coupling LBM and IB methods, the overheads of IB degrade the overall performance. As an alternative, we have explored different hybrid implementations that effectively hide such overheads and allow us to exploit both the multi-core and the hardware accelerator in a cooperative way, with excellent performance results.

5.1. Introduction

The dynamics of a solid in a flow field is a research topic with a growing interest in many scientific communities. It is intrinsically interdisciplinary (structural mechanics, fluid mechanics, applied mathematics, ...) and covers a broad range of applications (e.g. aeronautics, civil engineering, biological flows, etc.). The number of works in this field is rapidly increasing, which reflects the growing importance of studying the dynamics in the solid-fluid interaction [165, 24, 112, 84]. Most of these simulations are compute-intensive and benefit from high performance computing systems. However, they also exhibit an irregular and dynamic behaviour, which often leads to poor performance when using emerging heterogeneous systems equipped with many-core accelerators.

Many Computational Fluid Dynamic (CFD) applications and software packages have already been ported and redesigned to exploit heterogeneous systems. These developments have often involved major algorithm changes since some classical solvers may turned out to be inefficient or difficult to tune [152, 154]. Fortunately, other solvers are particularly well suited for GPU acceleration and are able to achieve significant performance improvements. The Lattice Boltzmann method (LBM) is one of those examples thanks to its inherently data-parallel nature. Cer-

tainly, the computing stages of LBM are amenable to fine grain parallelization in an almost straightforward way. This fundamental advantage of LBM has been consistently confirmed by many authors [15, 110, 167, 43], for a large variety of problems and computing platforms.

In this chapter we explore the benefits of LBM solvers on heterogeneous systems. Our target application is an integrated framework that uses the Immersed Boundary (IB) method to simulate the influence of a solid immersed in a incompressible flow [41]. Some recent works that cover subjects closely related with our contribution are [167] and [114]. In [167], authors presented an efficient 2D implementation of the LBM, which deals with geometries, by using curved boundaries-based methodologies, that is able to achieve a high performance on GPUs. Curved boundaries are taken into account via a non equilibrium extrapolation scheme developed in [59]. In [114], S. K. Layton et al. studied the solution of two-dimensional incompressible viscous flows with immersed boundaries using the *IB projection* method introduced in [142]. Their numerical framework is based on a Navier-Stokes solver and uses the `Cusp` library developed by Nvidia [33] for GPU acceleration. Our framework uses a different Immersed Boundary formulation based on the one introduced by Uhlmann [148], which is able to deal with complex, moving or deformable boundaries [102, 162, 71, 168, 169, 148, 3, 41]. Some previous performance results were presented in [153]. In this contribution we include a more elaborated discussion about performance results, and as a novelty, we explore alternative heterogeneous platforms based on the recently introduced Intel Xeon Phi device. Special emphasis is given to the implementation techniques adopted to mitigate the overhead of the immersed boundary correction and keep the solver highly efficient.

The remainder of this chapter is organized as follows. In Section 5.2 we introduce the physical problem at hand and the general numerical framework that has been selected to cope with it. In Section 5.3 we review some optimizations of the baseline global LBM solver. After that, we describe the different optimizations and parallel strategies envisaged to achieve high-performance when introducing the IB correction. In Section 5.4 we focus on optimising this correction as it was a standalone kernel. Additional optimisations that take into account the interaction with the global LBM solver are studied afterwards in Section 5.5. Finally, we discuss the performance results of the proposed techniques in Section 5.6. We conclude in Section 5.7 with a summary of the main contributions of this work.

5.2. Numerical framework

As mentioned above, we have explored in this work a numerical framework based on the Lattice Boltzmann method coupled to the Immersed Boundary method. This combination is highly attractive when dealing with immersed bodies for two main reasons: (1) the shape of the boundary, tracked by a set of Lagrangian nodes is a sufficient information to impose the boundary values; and (2) the force of the fluid on the immersed boundary is readily available and thus easily incorporated in the set of equations that govern the dynamics of the immersed object. In addition, it is also particularly well suited for parallel architectures, as the time advancement is explicit and most computations are local [41]. In what follows, we briefly recall the basic formulation of LBM and then we describe the investigated LBM-IB framework.

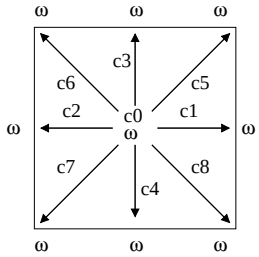


Figure 5.1: Standard two-dimensional 9-speed lattice (*D2Q9*) used in our work.

5.2.1. The LBM method

Lattice Boltzmann has been extensively used in the past decades (see [135] for a complete overview) and today is widely accepted in both academia and industry as a powerful and efficient alternative to classical Navier Stokes solvers for simulating (time-dependent) incompressible flows [43].

LBM is based on an equation that governs the evolution of a discrete distribution function $f_i(\mathbf{x},t)$ describing the probability of finding a particle at Lattice site $\mathbf{x}$ at time t with velocity $\mathbf{v} = \mathbf{e}_i$. In this work, we consider the *BGK* formulation [100] that relies upon an unique relaxation time τ toward the equilibrium distribution $f_i^{(eq)}$:

$$f_i\left(\mathbf{x}+\mathbf{e}_i\Delta t, t+\Delta t\right) - f_i\left(\mathbf{x},t\right) = -\frac{\Delta t}{\tau}\left(f\left(\mathbf{x},t\right) - f_i^{(eq)}\left(\mathbf{x},t\right)\right) + \Delta t F_i \tag{5.1}$$

The particles can move only along the links of a regular Lattice defined by the discrete speeds ($e_0 = c(0,0);\ e_i = c(\pm 1,0), c(0,\pm 1), i = 1\cdots 4;\ e_i = c(\pm 1,\pm 1)$, $c(\pm 1,\pm 1), i = 5\cdots 8$ with $c = \Delta x/\Delta t$) so that the synchronous particle displacements $\Delta\mathbf{x}_i = \mathbf{e}_i\Delta t$ never take the fluid particles away from the Lattice. For the present study, the standard two-dimensional 9-speed Lattice *D2Q9* (Figure 5.1) is used [5], but all the techniques that are presented in this work can be easily applied to three dimensional lattices.

The equilibrium function $f^{(eq)}(\mathbf{x},t)$ can be obtained by Taylor series expansion of the Maxwell-Boltzmann equilibrium distribution [107]:

$$f_i^{(eq)} = \rho\,\omega_i \left[1 + \frac{\mathbf{e_i}\cdot\mathbf{u}}{c_s^2} + \frac{(\mathbf{e_i}\cdot\mathbf{u})^2}{2c_s^4} - \frac{\mathbf{u}^2}{2c_s^2}\right] \tag{5.2}$$

In Equation 5.2, c_s is the speed of sound ($c_s = 1/\sqrt{3}$), ρ is the macroscopic density, and the weight coefficients ω_i are $\omega_0 = 4/9$, $\omega_i = 1/9$, $i = 1\cdots4$ and $\omega_5 = 1/36$, $i = 5\cdots8$ according to the current normalization. The macroscopic velocity $\mathbf{u}$ in Equation 5.2 must satisfy a Mach number requirement $|\,\mathbf{u}\,|/c_s \approx M << 1$. This stands as the equivalent of the CFL number for classical Navier Stokes solvers.

F_i in Equation 5.1 represents the contribution of external volume forces at lattice level that in our case include the effect of the immersed boundary. Given any external volume force $\mathbf{f}^{(ib)}(\mathbf{x},t)$, the contributions on the lattice are computed according to the formulation proposed in [59] as:

$$F_i = \left(1 - \frac{1}{2\tau}\right)\omega_i \left[\frac{\mathbf{e_i}-\mathbf{u}}{c_s^2} + \frac{\mathbf{e_i}\cdot\mathbf{u}}{c_s^4}\mathbf{e_i}\right]\cdot\mathbf{f_{ib}} \tag{5.3}$$

The multi-scale Chapman Enskog expansion of Equation 5.1, neglecting terms of $O(\varepsilon M^2)$ and using Equation 5.3, returns the Navier-Stokes equations with body forces and the kinematic viscosity related to lattice scaling as $\nu = c_s^2(\tau - 1/2)\Delta t$.

Without the contribution of the external volume forces stemming from the immersed boundary treatment, Equation 5.1 is advanced forward in time in two main stages, known as *collision* and *streaming*, as follows:

1. Calculation of the local macroscopic flow quantities ρ and $\mathbf{u}$ from the distribution functions $f_i(\mathbf{x},t)$:

$$\rho = \sum f_i(\mathbf{x},t) \text{ and } \rho\mathbf{u} = \sum \mathbf{e}_i f_i(\mathbf{x},t)$$

2. Collision:

$$f_i^* (\mathbf{x}, t+\Delta t) = f_i (\mathbf{x},t) - \tfrac{\Delta t}{\tau} \left(f(\mathbf{x},t) - f_i^{(eq)} (\mathbf{x},t) \right)$$

3. Streaming:

$$f_i (\mathbf{x} + \mathbf{e}_i \Delta t, t+\Delta t) = f_i^* (\mathbf{x}, t+\Delta t)$$

A large number of researchers have investigated the performance aspects of this update process [158, 13, 110, 61, 160, 73, 43] over the past decade. In Section 5.3 we review these previous works and describe the implementation techniques that we have opted to explore in our framework.

5.2.2. The LBM-IB framework

Next, we briefly introduce the Immersed Boundary method that we use both to enforce boundary values and to recover the fluid force exerted on immersed objects [148, 3]. In the selected Immersed Boundary approach (as in several others), the fluid is discretized on a regular Cartesian lattice while the immersed objects are discretized and tracked in a Lagrangian fashion by a set of markers distributed along their boundaries. The general setup of the investigated Lattice Boltzmann-Immersed Boundary method can be recast in the following algorithmic sketch.

1. Given $f_i(\mathbf{x},t)$ compute:

$$\rho = \sum f_i(\mathbf{x},t) \text{ and}$$

$$\rho \mathbf{u} = \sum \mathbf{e}_i f_i(\mathbf{x},t) + \frac{\Delta t}{2}\mathbf{f}_{ib}$$

2. Collision stage:

$$\hat{f}_i^* (\mathbf{x}, t+\Delta t) = f_i(\mathbf{x},t) - \frac{\Delta t}{\tau}\left(f(\mathbf{x},t) - f_i^{(eq)}(\mathbf{x},t)\right)$$

3. Streaming stage:

$$\hat{f}_i(\mathbf{x}+\mathbf{e}_i\Delta t, t+\Delta t) = \hat{f}_i^*(\mathbf{x}, t+\Delta t)$$

4. Compute:

$$\hat{\rho} = \sum \hat{f}_i(\mathbf{x}+\mathbf{e}_i\Delta t, t+\Delta t) \text{ and}$$
$$\hat{\rho}\hat{\mathbf{u}} = \sum \mathbf{e}_i \hat{f}_i(\mathbf{x}+\mathbf{e}_i\Delta t, t+\Delta t)$$

5. Interpolate on Lagrangian markers (volume force):

$$\hat{\mathbf{U}}(\mathbf{X}_k, t+\Delta t) = \mathscr{I}(\hat{\mathbf{u}}) \text{ and}$$
$$\mathbf{f}_{ib}(\mathbf{x},t) = \frac{1}{\Delta t}\mathscr{S}\left(\mathbf{U}_d(\mathbf{X}_k, t+\Delta t) - \hat{\mathbf{U}}(\mathbf{X}_k, t+\Delta t)\right)$$

6. Repeat collision with body forces (see 5.3) and Streaming:

$$f_i^*(\mathbf{x}, t+\Delta t) = f_i(\mathbf{x},t) - \frac{\Delta t}{\tau}\left(f(\mathbf{x},t) - f_i^{(eq)}(\mathbf{x},t)\right) + \Delta t F_i \text{ and}$$
$$f_i(\mathbf{x}+\mathbf{e}_i\Delta t, t+\Delta t) = f_i^*(\mathbf{x}, t+\Delta t)$$

As outlined above, the basic idea is to perform each time step twice. The first one, performed without body forces, predicts the velocity values at the immersed

boundary markers and the force distribution that restores the desired velocity boundary values at their locations. The second one applies the regularized set of singular forces and repeats the procedure (using Equation 5.3) to determine the final values of the distribution function at the next time step. A key aspect of this algorithm is the way by which the interpolation $\mathscr{I}$ and the $\mathscr{S}$ operators (termed as spread from now on) are applied. Here, following [148] and [3], we perform both operations (interpolation and spread) through a convolution with a compact support mollifier meant to mimic the action of a Dirac's delta. Combining the two operators we can write in a compact form:

$$\mathbf{f}_{(ib)}(\mathbf{x},t) = \frac{1}{\Delta t}\int_{\Gamma}\left(\mathbf{U_d}(\mathbf{s},t+\Delta t) - \int_{\Omega}\hat{\mathbf{u}}(\mathbf{y})\tilde{\delta}(\mathbf{y}-\mathbf{s})d\mathbf{y}\right)\tilde{\delta}(\mathbf{x}-\mathbf{s})d\mathbf{s} \tag{5.4}$$

where $\tilde{\delta}$ is the mollifier, Γ is the immersed boundary, Ω is the computational domain, and $\mathbf{U_d}$ is the desired value on the boundary at the next time step. The discrete equivalent of Equation 5.4 is simply obtained by any standard composite quadrature rule applied on the union of the supports associated to each Lagrangian marker. As an example, the quadrature needed to obtain the force distribution on the lattice nodes is given by:

$$f^l_{ib}(x_i,y_j) = \sum_{n=1}^{N_e} F^l_{ib}(\mathbf{X_n})\tilde{\delta}(x_i - X_n, y_j - Y_n)\varepsilon_n \tag{5.5}$$

where the superscript l refers to the l^{th} component of the immersed boundary force, (x_i,y_j) are the lattice nodes (*Cartesian* points) falling within the union of all the supports, N_e is the number of Lagrangian markers and ε_n is a value to be determined to enforce consistency between interpolation and the convolution (Equation 5.5).

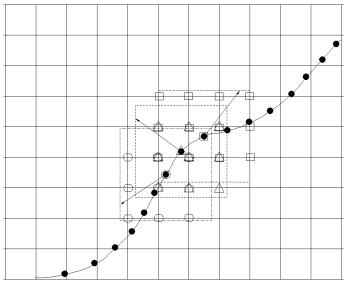


Figure 5.2: An immersed curve discretized with *Lagrangian* points (•). Three consecutive points are considered with the respective supports.

More details about the method in general and the determination of the ε_n values in particular can be found in [3].

In what follows we will give more details on the construction of the support cages surrounding each Lagrangian marker since it plays a key role in the parallel implementation of the IB algorithm. Figure 5.2 illustrates an example of the portion of the lattice units that falls within the union of all supports. As already mentioned, the embedded boundary curve is discretized into a number of markers $\mathbf{X}_I$, $I = 1..N_e$. Around each marker $\mathbf{X}_I$ we define a rectangular cage Ω_I with the following properties: (i) it must contain at least three nodes of the underlying Eulerian lattice for each direction; (ii) the number of nodes of the lattice contained in the cage must be minimized. The modified kernel, obtained as a Cartesian product of the one dimensional function [2]

$$\tilde{\delta}(r) = \begin{cases} \frac{1}{6}\left(5 - 3|r| - \sqrt{-3(1-|r|)^2+1}\right) & \\ & 0.5 \le |r| \le 1.5 \\ \frac{1}{3}\left(1 + \sqrt{-3r^2+1}\right) & 0.5|r| \le 0.5 \\ 0 & \text{otherwise} \end{cases} \tag{5.6}$$

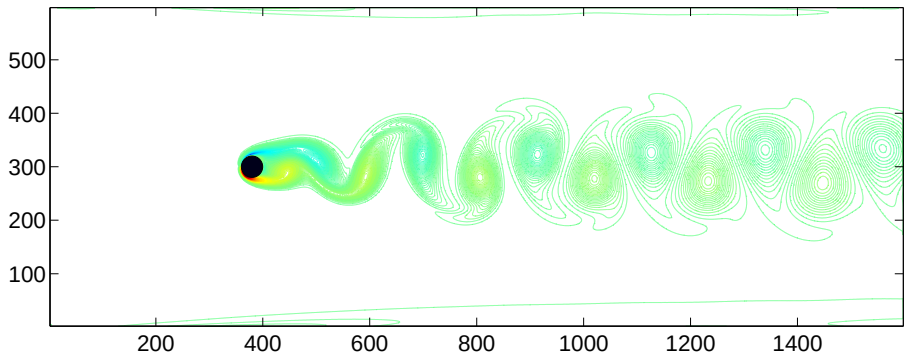


Figure 5.3: Vorticity with Re=100.

will be identically zero outside the square Ω_I. We take the edges of the square to measure slightly more than three lattice spacings Δ (i.e., the edge size is $3\Delta+\eta = 3+\eta$ in the actual LBM normalization). With such choice, at least three nodes of the lattice in each direction fall within the cage. Moreover a value of $\eta << 1$ ensures that the mollifier evaluated at all the nine (in two dimensions) lattice nodes takes on a non-zero value. The interpolation stage is performed locally on each nine points support: the values of velocity at the nodes within the support cage centered about each Lagrangian marker deliver approximate values (i.e., second order) of velocity at the marker location. The force spreading step requires information from all the markers, typically spaced $\Delta = 1$ apart along the immersed boundary. The collected values are then distributed on the union of the supports meaning that each support may receive information from supports centered about neighboring markers, as in Equation 5.5. The outlined method has been validated for several test cases including moving rigid immersed objects and flexible ones [41].

Author	**Re = 20**	**Re = 40**	**Re = 100**	
	C_D	C_D	C_D	C_L
Calhoun (2002)	2.19	1.62	1.33	0.298
Rusell (2003)	2.22	1.63	1.34	-
Silva (2003)	2.04	1.54	1.39	-
Xu (2006)	2.23	1.66	1.423	0.34
Zhou (2012)	2.3	1.7	1.428	0.315
This work	2.3	1.7	1.39	0.318

Table 5.1: Comparison between the numerical results yield by our method and previous studies.

5.2.3. Numerical validation of the method

Finally, we close this section presenting several test cases in order to validate the implementation of the code by comparing the numerical results obtained with other studies. One of the classical problems in CFD is the determination of the two-dimensional incompressible flow field around a circular cylinder, which is a fundamental problem in engineering applications. Several Reynolds numbers (20, 40 and 100) have been tested with the same configuration. The cylinder diameter D is equal to 40. The flow space is composed by a mesh equal to $40D$ (1600) $\times$ $15D$ (600). The boundary conditions are set as: Inlet: $\mathbf{u} = U, \mathbf{v} = 0$, Outlet: $\frac{\partial \mathbf{u}}{\partial x} = \frac{\partial \mathbf{v}}{\partial x} = 0$, Upper and lower boundaries: $\frac{\partial \mathbf{u}}{\partial y} = 0, \mathbf{v} = 0$, Cylinder surface: $\mathbf{u} = 0, \mathbf{v} = 0$, Volume fraction: 0.5236%. When Reynolds number is 20 and 40, there is no vortex structure formed during the evolution. The flow field is laminar and steady. In contrast, for the Reynolds number of 100, the symmetrical rectangular zones disappear and an asymmetric pattern is formed. The vorticity is shed behind the circular cylinder, and vortex structures are formed downstream. This phenomenon is graphically illustrated in Figure 5.3.

Two important dimensionless numbers are studied, the drag ($C_D = \frac{F_D}{0.5\rho U^2 D}$)

and lift ($C_L = \frac{F_L}{0.5\rho U^2 D}$) coefficients. F_D corresponds to the resistance force of the cylinder to the fluid in the streamwise direction and F_L is the lifting force of the circular cylinder, ρ is the density of the fluid, and U is the velocity of inflow. In order to verify the numerical results, the coefficients were calculated and compared with the results of previous studies (Table 5.1). The drag coefficient for Reynolds number of 20 and 40 is equal to the results presented by Zhou et al. [167]. The drag coefficient obtained for Reynolds number of 100 is identical to the results obtained by Silva et al. [84], and the lift coefficient is close to the presented by Zhou et al. [167].

5.3. Implementation of the global LBM update

5.3.1. Parallelization strategies

Parallelism is abundant in the LBM update and can be exploited in different ways. On our GPU implementation, the lattice nodes are distributed across GPU cores using a fine-grained distribution. As shown in Figure 5.4-bottom, we used a 1D Grid of 1D CUDA Block. Each CUDA-thread performs the LBM update of a single lattice node. On multi-core processors, cache locality is a major performance issue and it is better to distribute the lattice nodes across cores using a 1D coarse-grained distribution (Figure 5.4 top-left). The cache coherence protocol keeps the boundaries between subdomains updated. On the Intel Xeon Phi, we also distribute the lattice nodes across cores using a 1D coarse-grained distribution, but using a smaller block size (Figure 5.4 top-right).

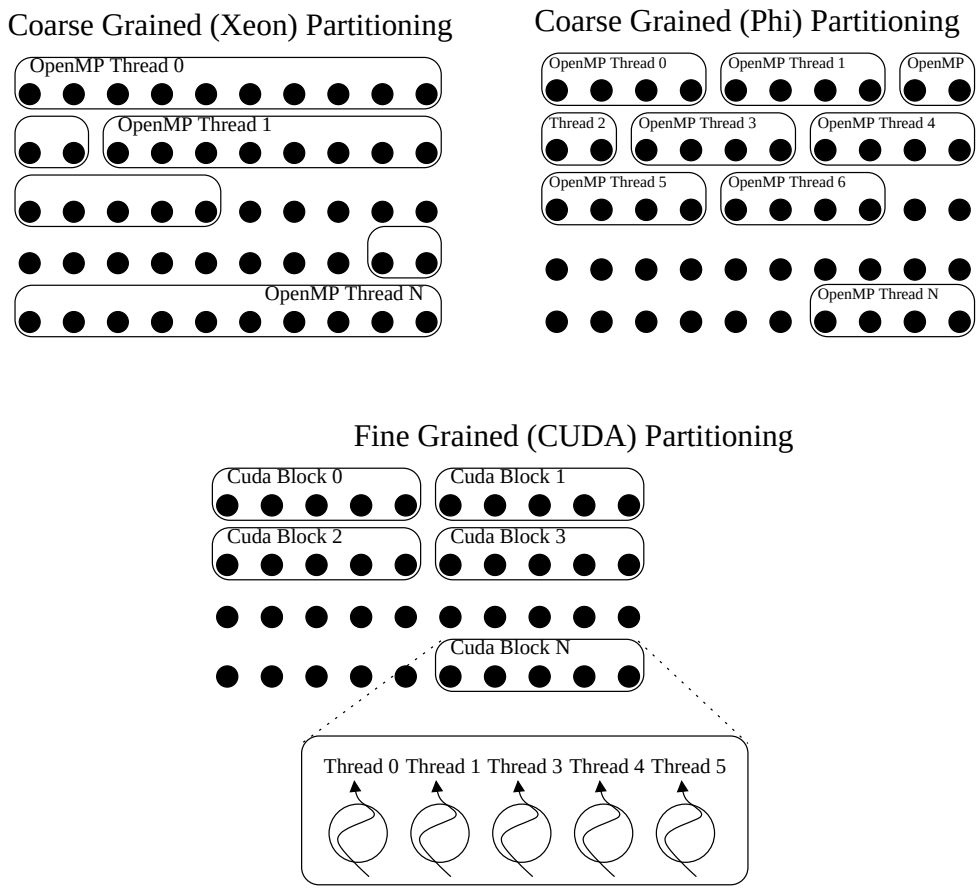


Figure 5.4: Fine-grained and coarse-grained distributions of the lattice notes.

Another important issue is how to implement a single LBM update. Conceptually, a discrete time step consists of a local collision operation followed by a streaming operation that propagates the new information to the neighbour lattice nodes. However, most implementations do not apply those operations as separate steps. Instead, they usually fuse in a single loop nest (that iterates over the entire domain), the application of both operations to improve temporal locality [158, 61].

This fused loop can be implemented in different ways. We have opted to use the pull scheme introduced in [158]. In this case, the body of the loop performs the streaming operation before collision, i.e. the distribution functions are gathered

(pulled) from the neighbours before computing the collision. Algorithm 5 shows a sketch of our implementation.

Algorithm 5 LBM *pull*.

1: Pull (f_1,f_2,ϖ,c_x,c_y)
2: x, y
3: x_{stream}, y_{stream}
4: $local_{u_x}$,$local_{u_y}$,$local_\rho$
5: $local_f[9], f_{eq}$
6: **for** $i = 1 \rightarrow 9$ **do**
7: $\quad x_{stream} = x - c_x[i]$
8: $\quad y_{stream} = y - c_y[i]$
9: $\quad local_f[i] = f_1[x_{stream}][y_{stream}][i]$
10: **end for**
11: **for** $i = 1 \rightarrow 9$ **do**
12: $\quad local_\rho += local_f[i]$
13: $\quad local_{u_x} += c_x[i] \times local_f[i]$
14: $\quad local_{u_y} += c_y[i] \times local_f[i]$
15: **end for**
16: $local_{u_x} = local_{u_x}/local_\rho$
17: $local_{u_y} = local_{u_y}/local_\rho$
18: **for** $i = 1 \rightarrow 9$ **do**
19: $\quad f_{eq} = \varpi[i] \cdot \rho \cdot (1 + 3 \cdot (c_x[i] \cdot local_{u_x} + c_y[i] \cdot local_{u_y}) + (c_x[i] \cdot local_{u_x} + c_y[i] \cdot local_{u_y})^2 - 1.5 \times ((local_{u_x})^2 + (local_{u_y})^2))$
20: $\quad f_2[x][y][i] = local_f[i] \cdot (1 - \frac{1}{\tau}) + f_{eq} \cdot \frac{1}{\tau}$
21: **end for**

Other implementations [167, 119, 15, 73] have used the traditional implementation sketched in Algorithm 6, which performs the collision operation before the streaming. It is known as the "push" scheme [158] since it loads the distribution function from the current lattice point and then it "pushes" (scatters) the updated values to its neighbours.

Algorithm 6 LBM *push*.

Collide Stream (u_x,u_y,ρ,f_1,f_2,ϖ,c_x,c_y)
x, y
x_{stream}, y_{stream}
f_{eq}
for $i = 1 \rightarrow 9$ **do**
    $f_{eq} = \varpi[i] \cdot \rho \cdot (1 + 3 \cdot (c_x[i] \cdot u_x[x][y] + c_y[i] \cdot u_y[x][y]) + (c_x[i] \cdot u_x[x][y] + c_y[i] \cdot u_y[x][y])^2 - 1.5 \times ((u_x[x][y])^2 + (u_y[x][y])^2))$
    $f_1[x][y][i] = f_1[x][y][i] \cdot (1 - \frac{1}{\tau}) + f_{eq} \cdot \frac{1}{\tau}$
    $x_{stream} = x + c_x[i]$
    $y_{stream} = y + c_y[i]$
    $f_2[x_{stream}][y_{stream}][i] = f_1[x][y][i]$
end for

Macroscopic (u_x,u_y,ρ,f_2,c_x,c_y)
x, y
$local_{u_x}$,$local_{u_y}$,$local_\rho$
for $i = 1 \rightarrow 9$ **do**
    $local_\rho += f_2[x][y][i]$
    $local_{u_x} += c_x[i] \times f_2[x][y][i]$
    $local_{u_y} += c_y[i] \times f_2[x][y][i]$
end for
$\rho[x][y] = local_\rho$
$u_x[x][y] = local_{u_x} / local_\rho$
$u_y[x][y] = local_{u_y} / local_\rho$

5.3.2. Data layout and memory management

Since LBM is a memory-bound algorithm, another important optimization problem is to maximize data locality. Many groups have considered this issue and have introduced several data layouts and code transformations that are able to improve locality on different architectures [158, 13, 61, 160, 43, 73, 126, 125]. Here we briefly describe the strategies used by our framework.

Different data structures have been proposed to store the discrete distribution functions f_i in memory:

- **AoS**. This data structure stores all the discrete distribution functions f_i of a given lattice point in adjacent memory positions (see Figure 5.5(a)). This way, it optimizes locality when computing the collision operation [125]. However, it does not provide good performance on GPU architectures since it leads to poor bandwidth utilization [15, 110, 73].

- **SoA**. In this alternative data structure, the discrete distribution functions f_i for a particular velocity direction are stored sequentially in the same array (see Figure 5.5(b)). Since each GPU thread handles the update of a single lattice node, consecutive GPU threads access adjacent memory locations with the SoA layout [15, 110, 73]. This way, they can be combined (coalesced) into a single memory transaction, which is not possible with the AoS counterpart.

- **SoAoS**. We have also explored a hybrid data structure, denoted as SoAoS in [125]. As SoA, it also allows coalesced memory transactions on GPUs. However, instead of storing the discrete distribution functions f_i for a particular velocity direction in a single sequential array, it distributes them across

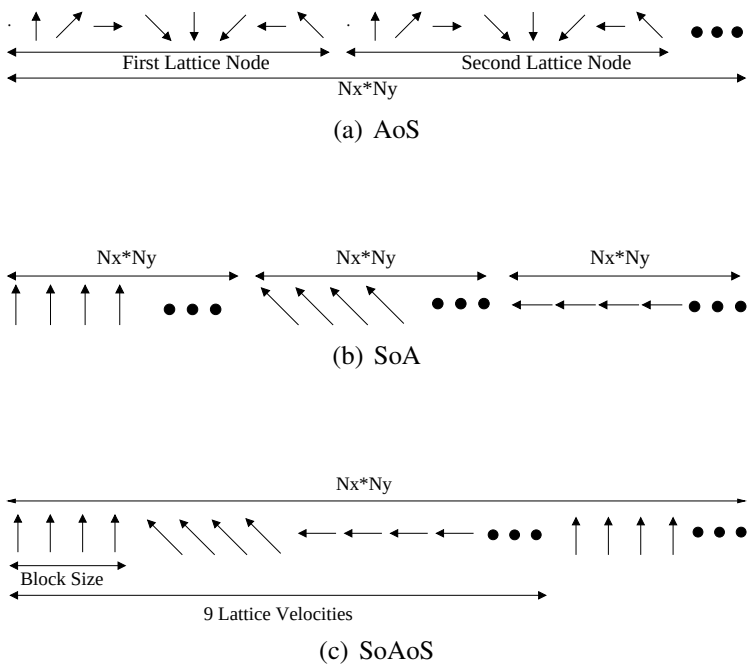


Figure 5.5: Different data layouts to store the discrete distribution functions f_i in memory.

different blocks of a certain block size (see Figure 5.5(c)). This size is a tunable parameter that trades off between spatial and temporal locality.

Apart from the the data layout, the memory management of the different implementations may also differ in the number of lattices that are used internally. We have used a two-lattice implementation, which is denoted as the AB scheme in [73, 13]. Essentially, AB holds the data of two successive time steps (A and B) and the simulation alternates between reading from A and writing to B, and vice versa. Other proposals, such as the AA data layout in [73, 13], are able to use a single copy of the distributions arrays and reduce the memory footprint. Some previous works have shown that those single lattice schemes outperform the AB scheme on multi-core processors (AA achieved the best results in [160]). However, on GPUs

the performance benefits of these schemes are less clear. In fact, a recent work has shown that both schemes get similar performance on GPUs [73] or the AB scheme is typically a little faster on the latest GPUs. On the other hand, AB simplifies the integration with IB correction, which is the main focus of this research, and is therefore clearly preferred in our framework.

5.4. Implementation of the IB correction

In this section we discuss the different strategies that we have explored to optimize and parallelize the IB correction, i.e. the computations related with the *Lagrangian* markers distributed on the solid(s) surface(s).

5.4.1. Parallelization strategies

The parallelization of this correction step is only effective as long as there is sufficient work, which depends on the number of *Lagrangian* points in the solid surface and the overlapping among their supports.

On multi-core processors, we have been able to achieve satisfactory efficiencies (even for moderate number of points) using a coarse-grained distribution of (adjacent) *Lagrangian* points across all cores. This implementation is relatively straightforward using a few OpenMP pragmas that annotate the loops that iterate over the Lagrangian points.

On GPUs, it is better to exploit fine-grained parallelism. Our implementation consists of two main kernels denoted as IF (*Immersed Forces*) and BF (*Body Forces*) respectively. Both of them use the 1D distribution of threads across Lagrangian points shown in Figure 5.6.

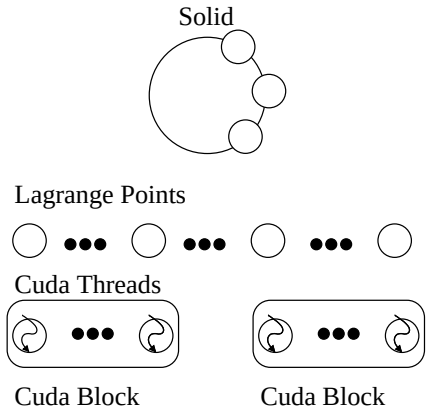


Figure 5.6: 1D fine-grained distribution of *Lagrangian* points across CUDA threads.

The IF kernel performs three major steps (see Algorithm 7). First, it assembles the velocity field on the supports. Then, it undertakes the interpolation at the *Lagrangian* markers and finally it determines the *Eulerian* volume force field on each node of the union of the supports. Note that forces are updated with *atomic* operations. Despite the overhead of such operations, they are necessary to prevent race conditions since the supports of different *Lagrangian* points can share the same *Eulerian* points, as graphically shown in Figure 5.2.

Once the IF kernel has been completed, the BF kernel (see Algorithm 8) computes the lattice forces and apply a local LBM update on the union of the supports, including now the contribution of the immersed boundary forces. Again, it avoids *race* conditions using *atomic* operations.

5.4.2. Data layout

Once again, another key aspect affecting performance in this correction step is the data structure (data layout) that is used to store the information about the *Lagrangian* points and their supports (coordinates, velocities and forces). Using the

Algorithm 7 Pseudo-code of the IF kernel

```
1: IFC_kernel(solid s,U_x,U_y)
2: vel_x, vel_y, force_x, force_y
3: for i = 1 → numSupport do
4:     vel_x += interpol(U_x[s.Xsupp[i]], s)
5:     vel_y += interpol(U_y[s.Ysupp[i]], s)
6: end for
7: force_x = computeForce(vel_x, s)
8: force_y = computeForce(vel_y, s)
9: for i = 1 → numSupport do
10:    AddAtom(s.XForceSupp, spread(force_x, s))
11:    AddAtom(s.YForceSupp, spread(force_y, s))
12: end for
```

Algorithm 8 Pseudo-code of the BF kernel

```
1: BFC_kernel(solid s,f_x,f_y)
2: F_body(Body Force),x,y,vel_x,vel_y
3: for i = 1 → numSupport do
4:     x = s.Xsupport[i]
5:     y = s.Ysupport[i]
6:     vel_x = s.VelXsupport[i]
7:     vel_y = s.VelYsupport[i]
8:     for j = 1 → 9 do
9:         F_body = (1 − 0.5 · 1/τ) · w[j] · (3 · ((c_x[j] − vel_x) · (f_x[x][y]) + (c_y[j] − vel_y) · f_y[x][y]))
10:        AddAtom(f^{n+1}[j][x][y], F_body)
11:    end for
12: end for
```

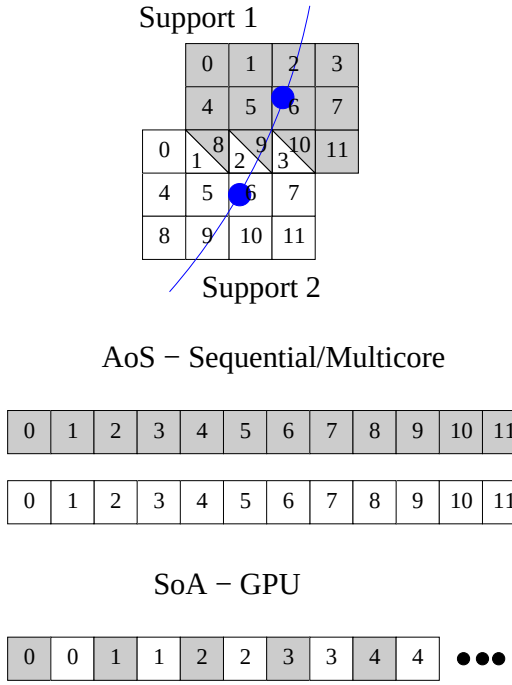


Figure 5.7: Different data layouts used to store the information about the coordinates, velocities and forces of the *Lagrangian* points and their supports. On multi-core processors we use an AoS data structure (top) whereas on GPUs (bottom) we use a SoA data structure.

locality principle, it is natural to use an array of structures (AoS) to place all this information at a given *Lagrangian* point in nearby memory locations. This is the data structure that we have used on multi-core processors, since it optimizes cache performance. In contrast, on GPUs it is more natural to use a SoA data structure that distributes the information of all *Lagrangian* points in a set of one-dimensional arrays. With SoA, consecutive threads access to contiguous memory locations and memory accesses are combined (coalesced) in a single memory transaction. Figure 5.7 illustrates the difference between those layouts in a simplified example with only two *Lagrangian* points.

5.5. Coupled LBM-IB on heterogeneous platforms

The complete LBM-IB framework can be fully implemented on manycore processors combining the strategies described earlier for the global LBM updated and the IB correction. In this approach, the host processor stays idle most of the time and it is used exclusively for:

- **Pre-processing**. The host processor sets up the initial configuration and uploads those initial data to the main memory of the accelerator.
- **Monitoring**. The host processor is also in charge of a monitoring stage that downloads the information of each lattice node (i.e., velocity components and density) back to the host main memory when required.

On GPUs, this approach consists of three main kernels, denoted in Figure 5.5 as LBM, IF and BF respectively, which are launched consecutively for every time step. The first kernel implements the LBM update while the other two perform the IB correction. The overhead of the pre-processing stage performed on the multi-core processor has been experimentally shown to be negligible and the data transfer of the monitoring stage are mostly overlapped with the execution of the LBM kernel.

Although this approach achieves satisfactory results on GPUs, its speedups are substantially lower than those achieved by pure LBM solvers [110, 73, 13, 43]. The obvious reason behind this behavior is the ratio between the characteristic volume fraction and the fluid field, which is typically very small. Therefore, the amount of data parallelism in the LBM kernel is substantially higher than in the other two kernels. In fact, for the target problems investigated, millions of threads compute the LBM kernel, while the IF and BF kernels only need thousands of them.

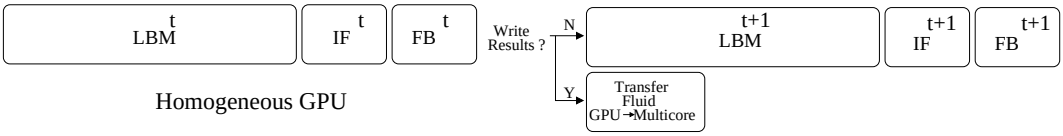


Figure 5.8: Homogeneous GPU Implementation. Both the LBM update and the IB correction are performed on the GPU. The host processors stays idle most of the time.

But in addition, those kernels also require *atomic* functions due to the intrinsic characteristics of the IB method, and those operations usually degrade performance.

5.5.1. LBM-IB on hybrid multicore-GPU platforms

As an alternative to mitigate the overheads caused by the IB correction, we have explored hybrid implementations that take advantage of the host Xeon processor to hide them.

In Figure 5.9 we show the hybrid implementation that we proposed on a previous chapter for heterogeneous multicore-GPU platforms. The LBM update is performed on the GPU as in the previous approach. However, IB and an additional local correction to LBM on the supports of the *Lagrangian* points are performed on the host processor. Both steps are coordinated using a pipeline. This way, we are able to overlap the prediction of the fluid field for the $t+1$ iteration with the correction of the IB method on the previous iteration t. This is possible at the expense of a local LBM computation of the"$t+1$" iteration on the multi-core and additional data transfers of the *supports* between the GPU and the host processor at each simulation step.

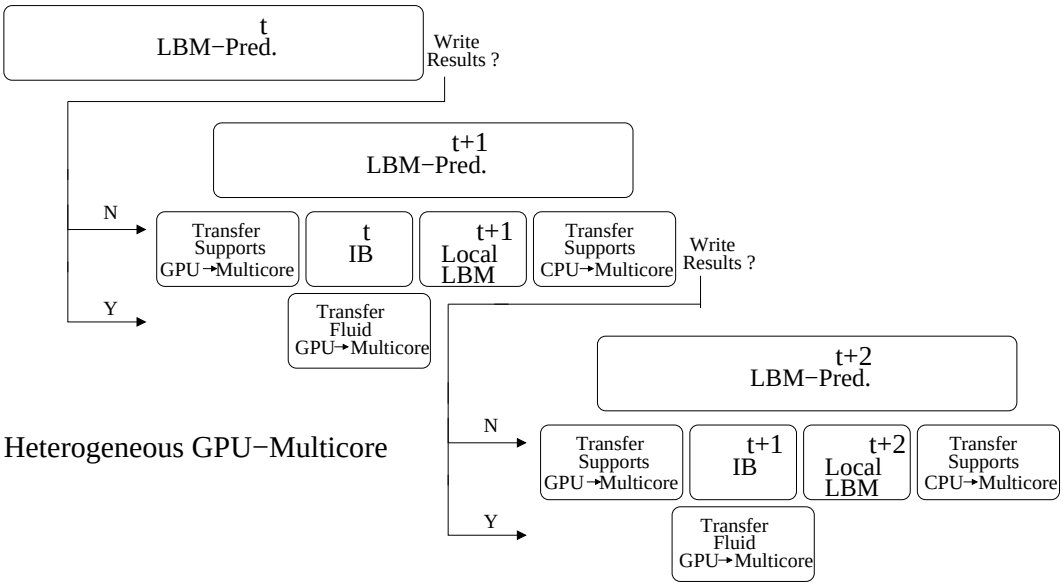


Figure 5.9: Hybrid multicore-GPU implementation.The LBM update is performed on the GPU, whereas the IB correction and an additional step to update the supports of the *Lagrangian* points are performed on the multi-core processor.

5.5.2. LBM-IB on hybrid multicore-Xeon Phi platforms

The homogeneous Xeon Phi implementation suffers from the same performance problems that the homogeneous GPU counterpart. But again, multicore-Xeon Phi collaboration allows us to achieve higher performance. It is possible to use the same hybrid strategy introduced above for the multicore-GPU platform. However, we have opted to use a slightly different implementation that simplifies code development. Figure 5.10 graphically illustrates the new partitioning. Essentially, it also consists on splitting the computational domain into two subdomains, so that one of them (*the solid subdomain*) fully includes the immersed solid. With such distribution, it is possible to perform the IB correction exclusively on the host Intel

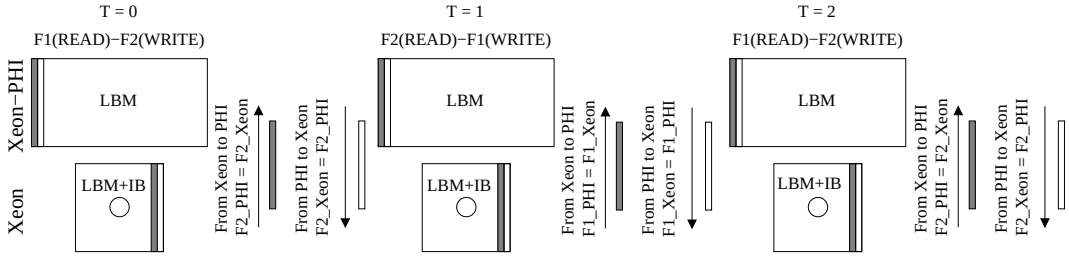


Figure 5.10: Hybrid multicore-Phi implementation. The $L_x \times L_y$ lattice is split into two 2D sub-domains so that the IB correction is only needed on one of the domains. The multi-core processor updates the $L_xIB \times L_y$ subdomain, which fully includes the immersed solid (marked as a circle). The Xeon Phi updates the the rest of the lattice nodes (the $L_xLBM \times L_y$ subdomain). The grey area highlights the ghost lattice nodes at the boundary between both sub-domains. In our target simulations, $L_xLBM \gg L_xIB$.

Xeon processor. However, in this case the host also performs the global LBM update on that subdomain, which simplifies the implementation.

As shown in Figure 5.10, every time step, it is necessary to exchange the boundaries between both subdomains. To reduce the penalty of such data transfers, we update the boundaries between subdomains at the beginning of each time step. With this transformation, it is possible to exchange those boundaries with asynchronous operations that are overlapped with the update of the rest of the subdomain. In our target simulations, the *solid subdomain* is much smaller that the Xeon Phi counterpart. To improve performance, the size of both subdomains is adjusted to balance the loads between both processors.

Platform Model	Intel Xeon E5520/E5-2670	NVIDIA GPU Geforce GTX780 (Kepler)	Intel Xeon Phi 5510P
Frequency Cores	2.26/2.6 GHz 8/16	0.863 2304	1.053 Ghz 60
On-chip Mem.	L1 32KB (per core) L2 512KB (unified) L3 20MB (unified)	SM 16/48KB (per MP) L1 48/16KB (per MP) L2 768KB (unified)	L1 32KB (per core) L2 256KB (per core) L2 30MB (coherent)
Memory Bandwidth	64/32GB DDR3 51.2 GB/s	6GB GDDR5 288 GB/s	8GB GDDR5 320 GB/s
Compiler	Intel Compiler 14.0.3	nvcc 6.5.12	Intel Compiler 14.0.3
Compiler Flags	-O3 -fomit-frame-pointer -fopenmp	-O3 -arch = sm_35	[1]

Table 5.2: Summary of the main features of the platforms used in our experimental evaluation.

5.6. Performance evaluation

5.6.1. Experimental setup

To critically evaluate the performance of the developed LBM-IB solvers, we have considered next a number of tests executed on two different heterogeneous platforms, whose main characteristics are summarized in Table 5.2.

On the GPU, the on-chip memory hierarchy has been configured as 16KB shared memory and 48KB L1, since our codes do not benefit from a higher amount of shared memory on the investigated tests.

Simulations have been performed using double precision, and we have used the conventional MFLUPS metric (millions of fluid lattice updates per second) to assess the performance.

5.6.2. Standalone Lattice-Boltzmann update

Before discussing the performance of our LBM-IB framework, it is important to determine the maximum performance that we can attain. We can estimate an upper limit omitting the IB correction. Figure 5.11 shows the performance of this benchmark on a Kepler GPU. Recall from Section 5.3 that our implementation is based

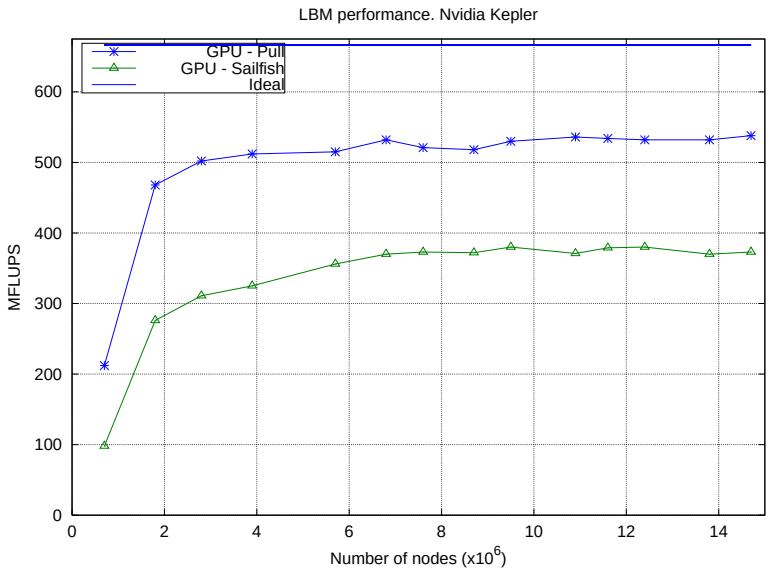


Figure 5.11: Performance of the LBM update on the NVIDIA Kepler GPU.

on the pull scheme and uses two lattice with the SoA data layout. As a reference we also show the performance of the *Sailfish* software package [73], which includes a LBM solver based on the *push* scheme, and the following estimation of the ideal MFLUPS [126]:

$$MFLUPS_{ideal} = \frac{B \times 10^9}{10^6 \times n \times 6 \times 8} \tag{5.7}$$

where $B \times 10^9$ is the memory bandwidth (GB/s), n depends on LBM model (DxQn), for our framework $n = 9$, D2Q9. The factor 6 is for the memory accesses, three read and write operations in the spreading step and three read and write operations in the collision step, and the factor 8 is for double precision (8 bytes).

Figures 5.12(a) and 5.12(b) shows the performance on an Intel Xeon server. Although on multi-core processor it is natural to use the *AoS* data layout, *SoA* and *SoAoS* (with a block size of 32 elements) turn to be the most efficient data layouts.

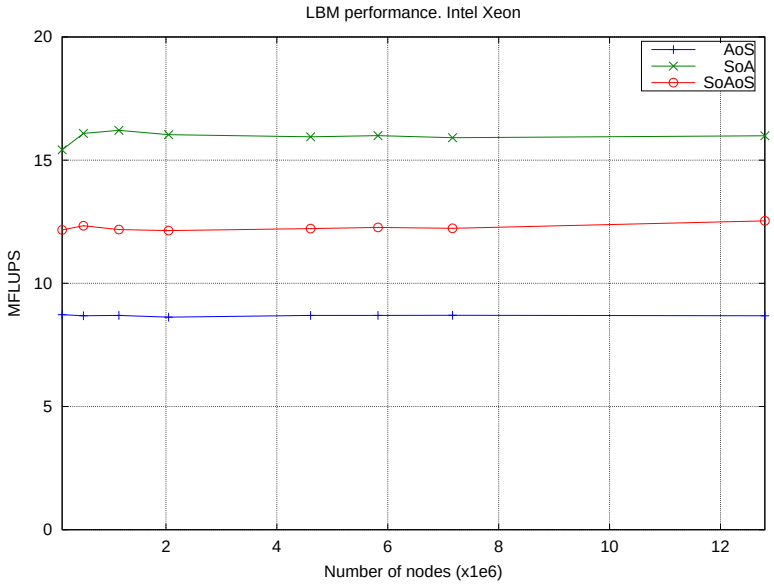
The main reason behind these unexpected results lies on the vector capabilities of modern processors. The compiler has been able to vectorize the main loops of the LBM update and both the AoS and SoAoS layouts allow a better exploitation of vectorization. Figure 5.13 shows the scalability on this solver. The observed speedup factors over the sequential implementation almost peak with 16 threads, but the application scales relatively well since its performance is limited by the memory bandwidth.

Figures 5.14(a) and 5.14(b) shows the performance on the Intel Xeon Phi. In this platform, *SoAoS* is able to outperform the other data layouts. The optimal SoAoS block size in this case is 128 elements, $4\times$ the block size of the Intel Xeon, which coincides with the ratio between the vector widths of both architectures.

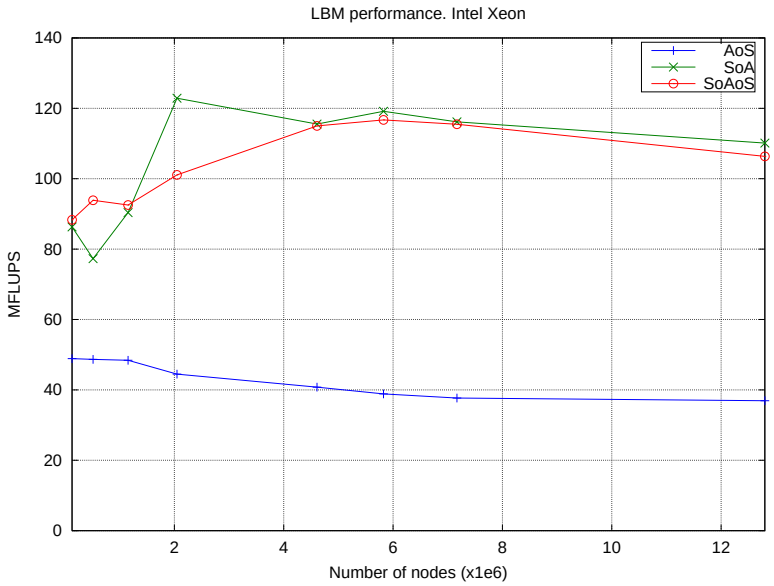
Figure 5.15 analyses the impact of the thread-core affinity on performance using three different pre-defined strategies (see [55] for details on the rationale of each strategy). The *compact* affinity provides the best performance although the differences between them are not significant.

Finally, Figure 5.16 shows the scalability on the Xeon Phi. For large problems, performance always improves as the number of threads increases. However, the gap with the ideal MFLUPS estimate is much larger in this case than in the other platforms. Given that the codes used for the Intel Xeon Phi and those used for the Intel Xeon are essentially the same (only optimal block sizes and minor optimization parameters change between both implementations) it is expected that either memory bandwidth is not limiting performance in this platform, or better performance values are expected for larger problem sizes.

Overall, the best performance is achieved on the GPU, which is able to outperform both the Intel Xeon and the Intel Xeon Phi. For the largest grid tested, the



(a) Sequential



(b) 32 threads

Figure 5.12: Performance of the LBM update on the Intel Xeon multi-core processor

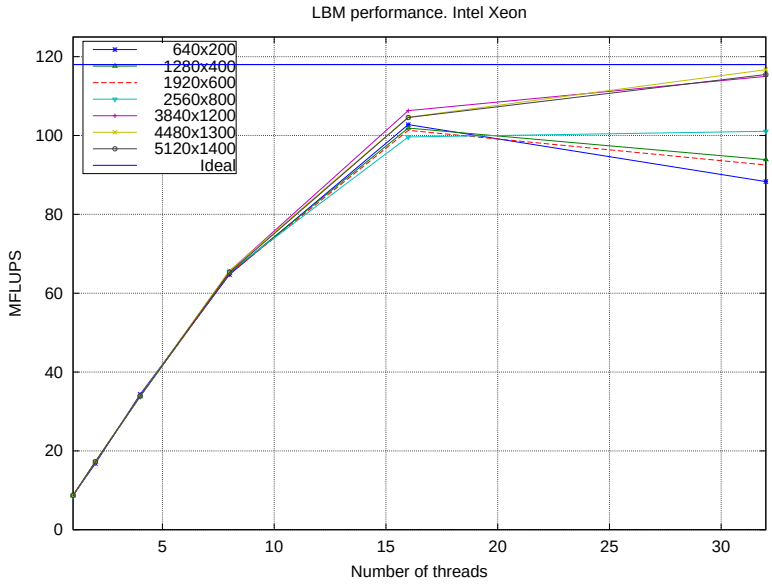
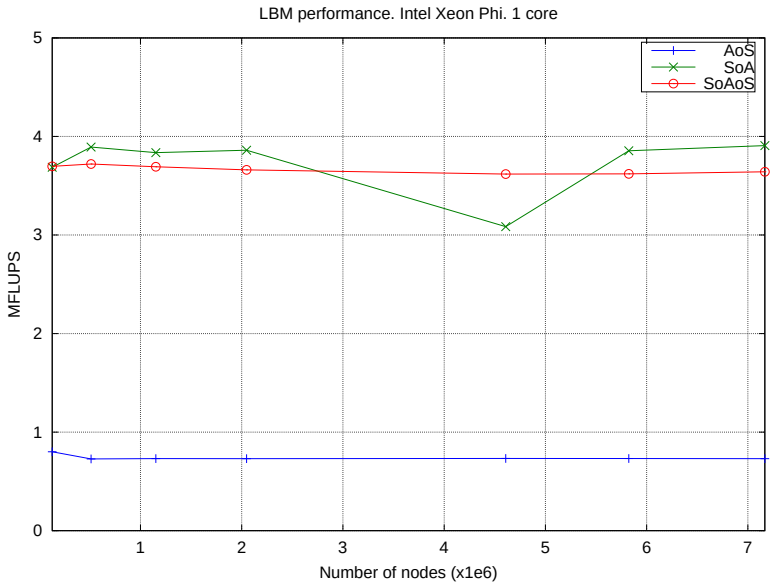


Figure 5.13: Scalability of the LBM update on an Intel Xeon multi-core processor.

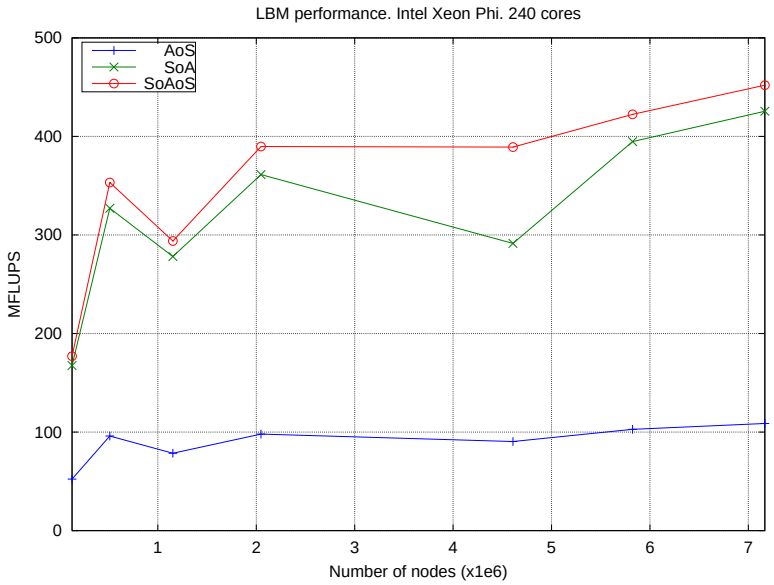
speed factor is 4.62× and 1.22×, respectively.

5.6.3. Standalone IB correction

Our second test uses a synthetic benchmark that focus exclusively on the IB correction, i.e., it omits the global LBM update and only applies the local IB correction. Figure 5.17 shows the speedups of the parallel implementations over a sequential counterpart for increasing number of *Lagrangian* nodes. We are able to achieve substantial speedups, even for a moderate number of nodes. Notably, the best performance is achieved on the GPU, despite the overheads caused by the atomic updates needed on that implementation. From 2500 *Lagrangian* markers, the GPU outperforms the 8-core multicore counterpart with a 2.5x speedup.



(a) Sequential



(b) 240 threads

Figure 5.14: Performance of the LBM update on the Intel Xeon Phi.

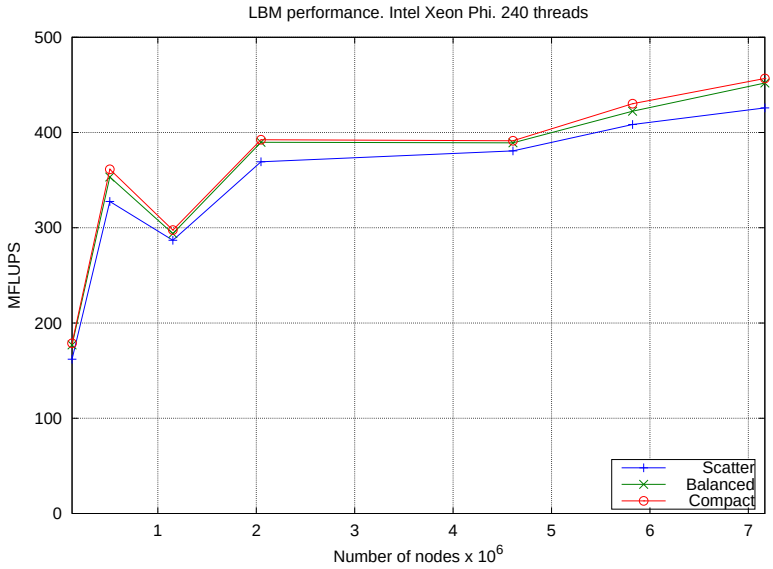


Figure 5.15: Performance of the LBM update with different thread-core affinity strategies on the Intel Xeon Phi.

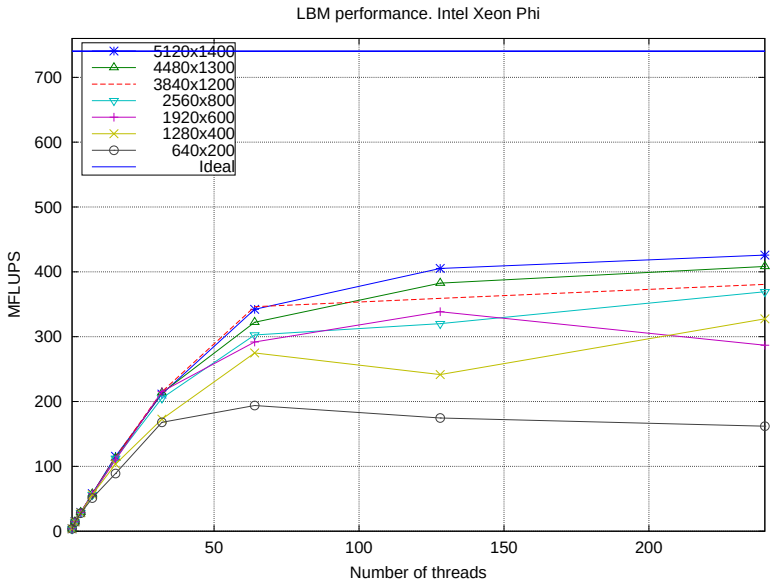


Figure 5.16: Scalability of the LBM update on an Intel Xeon Phi.

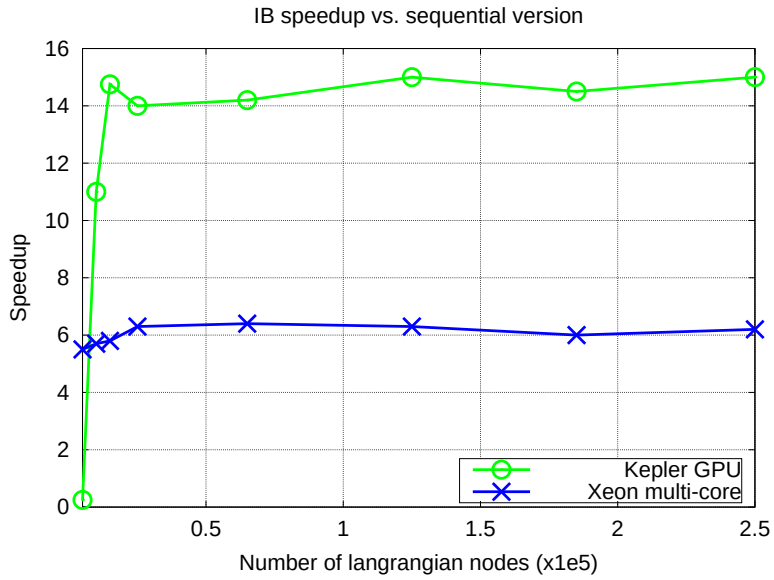


Figure 5.17: Speedups of the IB method on multi-core and GPU for increasing number of Lagrangian nodes.

5.6.4. Coupled LBM-IB on heterogeneous platforms

5.6.4.1. Multicore-GPU

Figure 5.18(a) shows the performance of the hybrid LBM-IB solver on a multicore-GPU heterogeneous platform for an increasing number of lattice nodes. As a reference, Figur 5.18(b) shows the performance of the homogeneous GPU implementation. We have used the same physical setting studied in Section 5.2 and we have investigated two realistic scenarios with characteristic volume fractions of 0.5% and 1% respectively (i.e. the amount of embedded *Lagrangian* markers also grows with the number of lattice nodes).

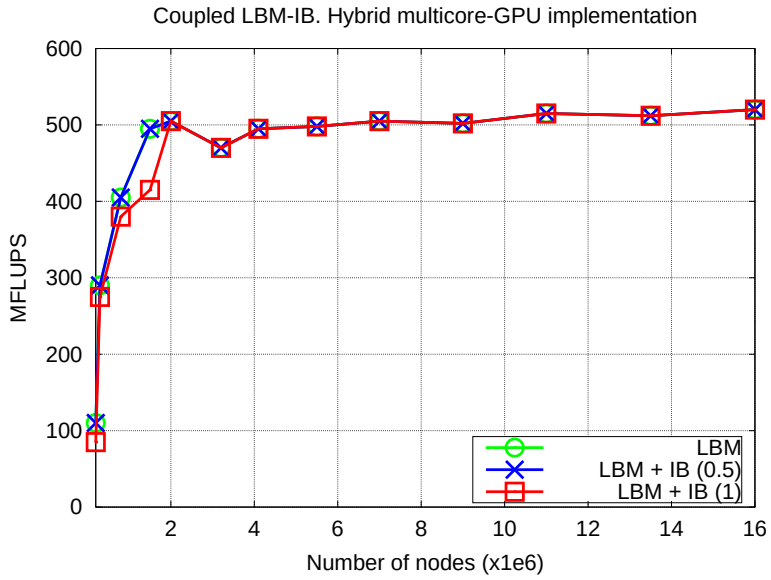
The performance (MFLUPS) of the homogeneous implementation (Figure 5.18(b)) drops substantially over the pure LBM implementation (Figure 5.11). The slow-down is around 15% for a solid volume fraction of 0.5%, growing to 25% for the

1% case. In contrast, for these fractions the hybrid GPU-multicore version is able to hide the overheads of the IB method, having similar performance to the pure LBM implementation.

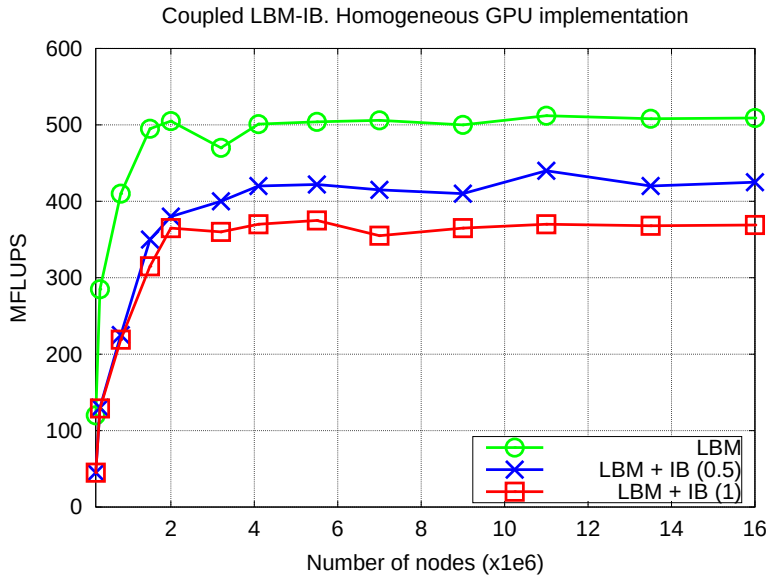
Figure 5.19 shows a breakdown of the execution time for the simulation with a solid volume fraction of 1%. On the hybrid implementation, the cost of the IB correction is higher than in the homogeneous counterpart, reaching around 65% of the total execution time. This is expected since the in this case, the IB correction includes additional data transfers and local LBM updates. However, these costs are hidden with the global LBM update.

5.6.4.2. Multicore-Xeon Phi

Similarly, the performance of the proposed strategy over the multicore-Xeon Phi heterogeneous platform is analyzed in Figure 5.20(a). We have focused on the same numerical scenario described earlier with a a solid volume fraction of 1%. As in the multicore-GPU platform, the observed performance in MFLUPS roughly matches the performance of the pure-LBM implementation on the same platform (Figure 5.14(b)), with a peak performance of 450 MFLUPS for the largest problem size. L_xIB, which defines the size of the subdomain that is simulated on the multicore processor, is a critical parameter in this implementation. As expected, there is an optimal value, which depends on the size of both the immersed solid and the grid, that balances the load in both processors. For example, the peak performance for the smallest grid tested ($L_x = 2560, L_y = 800$) is attained when the Xeon subdomain is roughly a 10% of the complete grid ($L_xIB = 250$), growing to 12% for the largest grid tested ($L_x = 5120, L_y = 1400$; ($L_xIB = 620$)).



(a) Hybrid multicore-GPU



(b) Homogeneous GPU

Figure 5.18: Performance of the complete LBM-IB solver for an increasing number of lattice nodes.

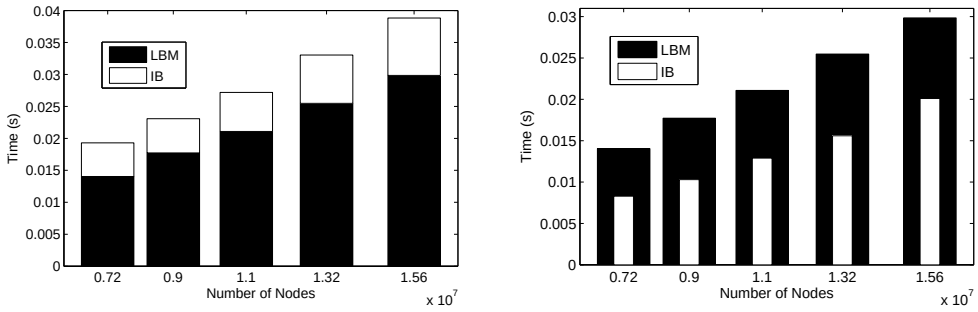


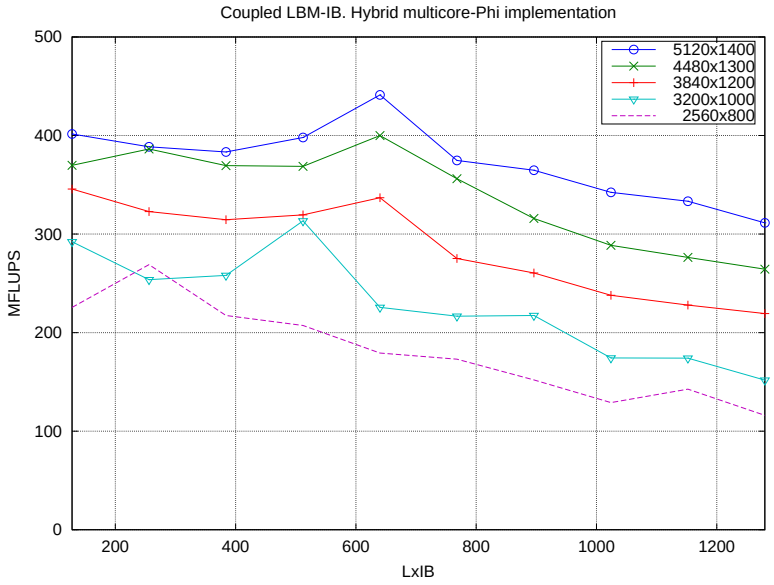
Figure 5.19: Execution time breakdown for a solid volume fraction of 1% of the LBM and IB kernels on the homogeneous GPU implementation (left) and multicore-GPU heterogeneous (right) platforms.

Overall, as we have aimed, the overhead of the solid interaction is mainly hidden thanks to the effective cooperation with the multi-core processor. The penalty of the data transfers between both processors is negligible since boundary exchanges are conveniently orchestrated to allow their overlapping with useful computations.

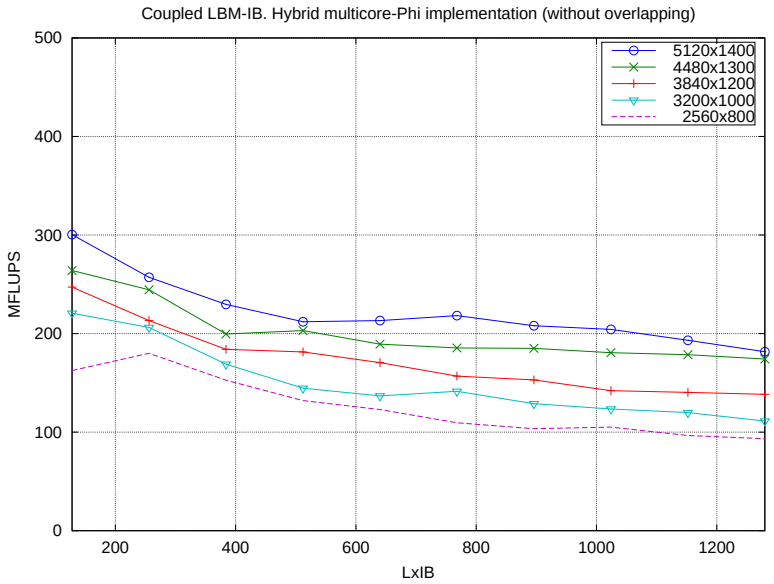
Finally, as a reference, Figure 5.20(b) shows the performance attained by an equivalent LBM-IB implementation, but without overlapping the execution on the Intel Xeon and the Intel Xeon Phi, i.e. the Xeon and the Phi subdomains are updated sequentially, one after the other. In this case, the peak performance drops to 300 MFLUPS, which highlights the benefit of overlapping the execution of both processors. Obviously, the lower L_xIB, the higher performance we achieve with this synthetic code since it increases the amount of work delivered to the Intel Xeon Phi.

5.7. Concluding remarks

In this final chapter, we have studied some approaches for increasing the performance of a coupled Lattice-Boltzmann and Immersed Boundary method to simulate Fluid-Solid interaction. We have focused on three state-of-the-art parallel systems:



(a) Hybrid multicore-Xeon Phi



(b) Hybrid multicore-Xeon Phi (without overlapping)

Figure 5.20: Performance of the complete LBM-IB solver on the multicore-Phi platform as the size of subdomain that is simulated on the multi-core procesor increases.

an Intel Xeon server, a NVIDIA KeplerGPU and an Intel Xeon Phi accelerator. Our baseline LBM solver uses most of the state-of-the-art code transformations that have been described in previous work. Notably, performance results (MFLUPS) on the GPU and the multicore processor are close to ideal MFLUPS estimations. On the Xeon Phi, the gap with these ideal estimations is higher, but it also achieves competitive performance. Overall, the best results are achieved on the GPU, which peaks at 550 MFLUPs, whereas the Intel Xeon Phi peaks at 450 MFLUPs.

Using the same design explored in the previous chapter, our hybrid implementation takes advantage of both the accelerators (GPU/Xeon Phi) and the multicore processors in a co-operative way to solve the coupled LBM-IB problem. For interesting physical scenarios with realistic solid volume fractions, the investigated hybrid solvers are able to hide the overheads caused by the IB correction and match the performance (in terms of MFLUPS) of state-of-the-art pure LBM solvers. This has been possible thanks to (1) the effective cooperation between the accelerator and the multicore processor and (2) the overlapping of data transfers between their memory spaces with useful computations.

Based on the presented LBM-IB framework, we envision two main research lines as future work. First, it will be interesting to study the adaptation of our framework to scenarios that deal with deformable and moving bodies, which require more dynamic work partitioning and assignment strategies. Second, we also plan to generalize our work to 3D simulations; for these problems, memory consumption arises as one of the main problems, naturally driving to distributed-memory architectures, possibly equipped with hardware accelerators. In this case, we believe many of the techniques presented in this work will be of direct application at each node level; the scalability and parallelization strategies across nodes is still a topic

under study.

Bibliography

[1] The imagine stream processor. In *Proceedings of the 2002 IEEE International Conference on Computer Design: VLSI in Computers and Processors (ICCD'02)*, ICCD '02, pages 282–, Washington, DC, USA, 2002. IEEE Computer Society.

[2] C. S. Peskin A. M. Roma and M. J. Berger. An adaptive version of the immersed boundary method. *Journal of Computational Physics*, 153:509 – 534, 1999.

[3] U. Piomelli A. Pinelli, I. Naqavi and J. Favier. Immersed-boundary methods for general finite-differences and finite-volume navier-stokes solvers. *Journal of Computational Physics*, 229(24):9073 – 9091, 2010.

[4] M.J. Aftosmis, M.J. Berger, and G. Adomavicius. A parallel cartesian approach for external aerodynamics of vehicles with complex geometry. *Proceedings of the Thermal and Fluids Analysis Workshop, NASA Marshall Spaceflight Center, Huntsville, AL*, 1999.

[5] Cyrus K. Aidun and Jonathan R. Clausen. Lattice-boltzmann method for complex flows. *Annual Review of Fluid Mechanics*, 42(1):439–472, 2010.

[6] G. Alverson, R. Alverson, D. Callahan, B. Koblenz, A. Porterfield, and B. Smith. The tera computer system. *In Proceesing of the Fourth international Conference on Supercomputing*, pages 1–6, 1990.

[7] G. Alverson, R. Alverson, D. Callahan, B. Koblenz, A. Porterfield, and B. Smith. Exploiting heterogeneos parallelism on a multithreading multiproccesor. *In Proceesing of the Sixth international Conference on Supercomputing*, pages 188–197, 1992.

[8] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, AFIPS '67 (Spring), pages 483–485, New York, NY, USA, 1967. ACM.

[9] Pierluigi Amodio and Francesca Mazzia. A parallel gauss-seidel method for block tridiagonal linear systems. *SIAM J. Sci. Comput.*, 16(6):1451–1461, November 1995.

[10] J. D. Anderson. *Computational Fluid Dynamics - The Basics with Applications*, volume 1. McGraw-Hill, 1995.

[11] J. D. Anderson, F. R. Menter, E. Dick, G. Degrez, and J. Vierendeels. *Introduction to Computational Fluid Dynamics*. von Karman Institute for Fluid Dynamics, 2013.

[12] Lilit Axner, Alfons G. Hoekstra, Adam Jeays, Pat Lawford, Rod Hose, and Peter MA Sloot. Simulations of time harmonic blood flow in the mesenteric artery: comparing finite element and lattice boltzmann methods. *BioMedical Engineering OnLine*, 2000.

[13] P. Bailey, J. Myre, S.D.C. Walsh, D.J. Lilja, and M.O. Saar. Accelerating lattice boltzmann fluid flow simulations using graphics processors. In *Parallel Processing, 2009. ICPP '09. International Conference on*, pages 550–557, Sept 2009.

[14] G.H. Barnes, R.M. Brown, M. Kato, David J. Kuck, D.L. Slotnick, and R.A. Stokes. The illiac iv computer. *Computers, IEEE Transactions on*, C-17(8):746–757, Aug 1968.

[15] M. Bernaschi, M. Fatica, S. Melchiona, S. Succi, and E. Kaxiras. A flexible high-performance lattice boltzmann gpu code for the simulations of fluid flows in complex geometries. *Concurrency Computa.: Pract. Exper.*, 22:1–14, 2010.

[16] M. Bernaschi, M. Fatica, S. Melchionna, S. Succi, and E. Kaxiras. A flexible high-performance lattice boltzmann gpu code for the simulations of fluid flows in complex geometries. *Concurrency and Computation: Practice and Experience*, 22(1):1–14, 2010.

[17] Hester Bijl and Pieter Wesseling. A unified method for computing incompressible and compressible flows in boundary-fitted coordinates. *Journal of Computational Physics*, 141(2):153 – 173, 1998.

[18] DarioA. Bini and Beatrice Meini. The cyclic reduction algorithm: from poisson equation to stochastic processes and beyond. *Numerical Algorithms*, 51(1):23–60, 2009.

[19] F. G. Blottner. Chemical nonequilibrium boundary layer. *AIAA Journal*, 2(2):232–239, 1964.

[20] F. G. Blottner. Nonequilibrium laminar boundary-layer flow of ionized air. *AIAA Journal*, 2(11):1921–1927, 1964.

[21] A. Branover, D. Foley, and M. Steinman. Amd fusion apu: Llano. *Micro, IEEE*, 32(2):28–37, March 2012.

[22] R. Buchty, V. Heuveline, W. K., and J.-P. Weiss. A direct method for the discrete solution of separable elliptic equations. *Concurrency and Computation: Practice and Experience*, 24(7):663–675, 2012.

[23] O. Buneman. A compact non-iterative Poisson solver. Technical Report 294, Institute for Plasma Research, Stanford University, Stanford, CA, USA, 1969.

[24] Donna Calhoun. A cartesian grid method for solving the two-dimensional streamfunction-vorticity equations in irregular regions. *Journal of Computational Physics*, 176(2):231 – 275, 2002.

[25] Tuncer Cebeci, Jian P. Shao, Fassi Kafyeke, and Eric Laurendeau. *Computational Fluid Dynamics for Engineers*. Springer, Inc., 2005.

[26] S. Chapman and T. G. Cowling. *The mathematical theory of non-uniform gases*, volume 3. Cambridge University Press, 1991.

[27] J. Choi, J. Demmel, I. Dhillon, J. Dongarra, S. Ostrouchov, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. ScaLAPACK: A portable linear algebra library for distributed memory computers — Design issues and performance. In Jack Dongarra, Kaj Madsen, and Jerzy Waśniewski, editors, *Applied Parallel Computing Computations in Physics, Chemistry and Engineering Science*, volume 1041 of *Lecture Notes in Computer Science*, pages 95–106. Springer Berlin Heidelberg, 1996.

[28] NVIDIA Corporation. Nvidia cuda c programming best practices guide. 2014.

[29] G. Crimi, F. Mantovani, M. Pivanti, S.F. Schifano, and R. Tripiccione. Early experience on porting and running a lattice boltzmann code on the xeon-phi co-processor. *Procedia Computer Science*, 18(0):551 – 560, 2013. 2013 International Conference on Computational Science.

[30] cuFFT. http://developer.nvidia.com/cuda/cufft.

[31] David E. Culler, Anoop Gupta, and Jaswinder Pal Singh. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 1997.

[32] W. J. Dally, F. Labonte, A. Das, P. Hanrahan, J. Ahn, J. Gummaraju, M. Erez, N. Jayasena, I. Buck, T. J. Knight, and U. J. Kapasi. Supercomputing with streams. *In Proceesing of the 2003 ACM/IEEE Conference on Supercomputing*, 2003.

[33] S. Dalton, N. Bell, L. Olson, and Michael Garland. Cusp: Generic parallel algorithms for sparse matrix and graph computations. *Available at http://cusplibrary.github.io/*, 2014.

[34] Andrew Davidson, Yao Zhang, and John D. Owens. An auto-tuned method for solving large tridiagonal systems on the gpu. In *Proceedings of the 25th IEEE International Parallel and Distributed Processing Symposium*. IEEE, IEEE, May 2011.

[35] R.H. Dennard, F.H. Gaensslen, V.L. Rideout, E. Bassous, and A.R. LeBlanc. Design of ion-implanted mosfet's with very small physical dimensions. *Solid-State Circuits, IEEE Journal of*, 9(5):256–268, October 1974.

[36] Peter J. Denning and Jack B. Dennis. The resurgence of parallelism. *Commun. ACM*, 53(6):30–32, June 2010.

[37] E. Dick. Introduction to finite element methods in computational fluid dynamics. In JohnF. Wendt, editor, *Computational Fluid Dynamics*, pages 235–274. Springer Berlin Heidelberg, 2009.

[38] G. Doolen. *Lattice gas methods for partial differential equations*, volume 4. Addison-Wesley, 1990.

[39] Hadi Esmaeilzadeh, Emily Blem, Renee St. Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark silicon and the end of multicore scaling. In *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ISCA '11, pages 365–376, New York, NY, USA, 2011. ACM.

[40] Rüdiger Esser and Renate Knecht. Intel paragon xp/s - architecture and software environment. In Hans-Werner Meuer, editor, *Supercomputer '93*, Informatik aktuell, pages 121–141. Springer Berlin Heidelberg, 1993.

[41] Julien Favier, Alistair Revell, and Alfredo Pinelli. A lattice boltzmann–immersed boundary method to simulate the fluid interaction with

moving and slender flexible objects. *Journal of Computational Physics*, 261(0):145 – 161, 2014.

[42] J. A. Fay and F. R. Riddell. Theory of stagnation point heat transfer in dissociated air. *Journal of the Aeronautical Sciences*, 25(2):73–85, 1958.

[43] Christian Feichtinger, Johannes Habich, Harald Köstler, Ulrich Rude, and Takayuki Aoki. Performance modeling and analysis of heterogeneous lattice boltzmann simulations on cpu–gpu clusters. *Parallel Computing*, 46(0):1 – 13, 2015.

[44] W.-C. Feng and D. Manocha. High-performance computing using accelerators. *Parallel Computing*, 33:645 – 647, 2007.

[45] J. H. Ferziger and M. Perîc. *Computational Methods for Fluid Dynamics*. Springer, Inc., 2002.

[46] 3D FFT. http://charm.cs.uiuc.edu/cs498lvk/projects/kunzman/index.htm.

[47] Fast Fourier Transform Using OpenMP FFT OpenMP. http://people.sc.fsu.edu/jburkardt/c_src/fft_openmp/fft_openmp.html.

[48] FISHPACK. http://www.cisl.ucar.edu/css/software/fishpack/.

[49] The International Technology Roadmap for Semiconductors. 2013 edition. Technical report, ITRS, 2013.

[50] U. Frisch, B. Hasslacher, and Y. Pomeau. Lattice-gas automata for the navier-stokes equation. *Phys. Rev. Lett.*, 56:1505–1508, 1986.

[51] Michael Garland and David B. Kirk. Understanding throughput-oriented architectures. *Commun. ACM*, 53(11):58–66, November 2010.

[52] David Geer. Industry trends: Chip makers turn to multicore processors. *Computer*, 38(5):11–13, 2005.

[53] Dominik Göddeke and Robert Strzodka. Cyclic reduction tridiagonal solvers on GPUs applied to mixed precision multigrid. *IEEE Transactions on Parallel and Distributed Systems (TPDS), Special Issue: High Performance Computing with Accelerators*, 22(1):22–32, January 2011.

[54] R. A. Graves. Computational fluid dynamics: The coming revolution. *Astronautics and Aeronautics*, 20(3):20–28, 1982.

[55] Ronald W Green. OpenMP thread affinity control. 2014.

[56] B.E. Griffith and C.S. Peskin. On the order of accuracy of the immersed boundary method: Higher order convergence rates for sufficient smooth problems. *Journal of Computational Physics*, 208:75 – 105, 2005.

[57] Johan Gronqvist and Anton Lokhmotov. Optimising opencl kernels for the arm mali tm-t600 gpus. In Wolfgang Engel, editor, *GPU Pro 5: Advanced Rendering Techniques*. CRC Press, 2014.

[58] JL Guermond, Peter Minev, and Jie Shen. An overview of projection methods for incompressible flows. *Computer methods in applied mechanics and engineering*, 195(44):6011–6045, 2006.

[59] Zhaoli Guo, Chuguang Zheng, and Baochang Shi. An extrapolation method for boundary conditions in lattice boltzmann method. *Physics of Fluids (1994-present)*, 14(6):2007–2010, 2002.

[60] A. Q. Eschenroeder H. G. Hall and P. V. Marrone. Blunt-nose inviscid airflows with coupled nonequilibrium processes. *Journal of the Aerospace Sciences*, 29(9):1038–1051, 1962.

[61] J. Habich, C. Feichtinger, H. Köstler, G. Hager, and G. Wellein. Performance engineering for the lattice boltzmann method on gpgpus: Architectural requirements and performance results. *Computers & Fluids*, 80(0):276 – 282, 2013. Selected contributions of the 23rd International Conference on Parallel Fluid Dynamics ParCFD2011.

[62] Wolfgang Hackbusch. *Multi-Grid Methods and Applications*, volume 1. Springer Berlin Heidelberg, 2003.

[63] Xiaoyi He and Li-Shi Luo. A priori derivation of the lattice boltzmann equation. *Phys. Rev. E*, 55:R6333–R6336, Jun 1997.

[64] John L. Hennessy and David A. Patterson. *Computer Architecture, Fifth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 5th edition, 2011.

[65] Jae Heon Yun. Parallel performance of block ilu preconditioners for a block-tridiagonal matrix. *The Journal of Supercomputing*, 24(1):69–89, 2003.

[66] F. J. Higuera and J. Jiménez. Boltzmann approach to lattice gas simulations. *Europhys. Lett.*, 9:663–668, 1989.

[67] F. J. Higuera, S. Succi, and R. Benzi. Lattice gas dynamics with enhanced collisions. *Europhys. Lett.*, 9:345–349, 1989.

[68] Steven P. Hirshman, Kalyan S. Perumalla, Vickie E. Lynch, and Raul Sanchez. Bcyclic: A parallel block tridiagonal matrix cyclic solver. *J. Comput. Physics*, 229(18):6392–6404, 2010.

[69] R. W. Hockney. A fast direct solution of poisson's equation using fourier analysis. *Journal of the ACM*, 12:95–113, 1965.

[70] R. W. Hockney and C. R. Jesshope. *Parallel Computers*, volume 1. Adam Hilger, Bristol, 1981.

[71] W.-X. Huang, S. J. Shin, and H. J. Sung. Simulation of flexible filaments in a uniform flow by the immersed boundary method. *J. Comput. Phys.*, 226(2):2206–2228, 2007.

[72] Intel. Intel 5520 chipset and intel 5500 chipset-datasheet. *Intel Corp.*, 2009.

[73] M. Januszewski and M. Kostur. Sailfish: A flexible multi-GPU implementation of the lattice Boltzmann method. *Computer Physics Communications*, 185(9):2350–2368, September 2014.

[74] Brian Jeff. Big.LITTLE system architecture from ARM: saving power through heterogeneous multiprocessing and task context migration. In Patrick Groeneveld, Donatella Sciuto, and Soha Hassoun, editors, *DAC*, pages 1143–1146. ACM, 2012.

[75] James Jeffers and James Reinders. *Intel Xeon Phi Coprocessor High Performance Programming*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2013.

[76] H. Ji, F.-S. Lien, and F. Zhang. A gpu-accelerated adaptive mesh refinement for immersed boundary methods. *Computers and Fluids (under revision)*.

[77] H. Ji, F.S. Lien, and E. Yee. An efficient second-order accurate cut-cell method for solving the variable coefficient poisson equation with jump conditions on irregular domains. *Int. J. Numer. Methods Fluids*, 52(7):723–748, 2006.

[78] D. Kandhai, D. J.-E. Vidal, A. G. Hoekstra, H. Hoefsloot, P. Iedema, and P. M. A. Sloot. A comparison between lattice-boltzmann and finite-element simulations of fluid flow in static mixer reactors. *International Journal of Modern Physics C*, 09(08):1123–1128, 1998.

[79] B.K. Khailany, T. Williams, J. Lin, E.P. Long, M. Rygh, D.W. Tovey, and W.J. Dally. A programmable 512 gops stream processor for signal, image,

and video processing. *Solid-State Circuits, IEEE Journal of*, 43(1):202–213, Jan 2008.

[80] Hee-Seok Kim, Shengzhao Wu, Li wen Chang, and Wen mei W. Hwu. A scalable tridiagonal solver for gpus. *2013 42nd International Conference on Parallel Processing*, 0:444–453, 2011.

[81] S. Kollmannsberger, S. Geller, A. Düster, J. Tölke, C. Sorger, M. Krafczyk, and E. Rank. Fixed-grid fluid–structure interaction in two dimensions based on a partitioned lattice boltzmann and p-fem approach. *International Journal for Numerical Methods in Engineering*, 79(7):817–845, 2009.

[82] Poonacha Kongetira, K. Aingaran, and K. Olukotun. Niagara: a 32-way multithreaded sparc processor. *Micro, IEEE*, 25(2):21–29, March 2005.

[83] Z. Kopal. Tables of supersonic flow around cones. *Dept. of electrical Engineering, Center of Analysis, massachusetts Institute of Technology, Cambridge*, 1947.

[84] A.L.F. Lima E Silva, A. Silveira-Neto, and J. J. R. Damasceno. Numerical simulation of two-dimensional flows over a circular cylinder using the immersed boundary method. *J. Comput. Phys.*, 189(2):351–370, August 2003.

[85] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym. Nvidia tesla: A unified graphics and computing architecture. *Micro, IEEE*, 28(2):39–55, March 2008.

[86] O. Malaspinas and P. Sagaut. Consistent subgrid scale modelling for lattice boltzmann methods. *Journal of Fluid Mechanics*, 700:514–542, 2012.

[87] Simon Marié, Denis Ricot, and Pierre Sagaut. Comparison between lattice boltzmann method and navier-stokes high order schemes for computational aeroacoustics. *J. Comput. Phys.*, 228(4):1056–1070, March 2009.

[88] M. F. McCracken and C.S. Peskin. A vortex method for blood flow through heart valves. *J. Comput. Phys.*, 35:183–205, 1980.

[89] Hans Werner Meuer and Horst Gietl. Supercomputers - prestige objects or crucial tools for science and industry? *Praxis der Informationsverarbeitung und Kommunikation*, 36(2):117–128, 2013.

[90] A. A. Mohamad. *The Lattice Boltzmann Method - Fundamental and Engineering Applications with Computer Codes*. Springer, 2011.

[91] Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8):56–59, April 1965.

[92] Mario Nemirovsky and Dean M. Tullsen. *Multithreading Architecture*. Synthesis Lectures on Computer Architecture. Morgan and Claypool Publishers, 2013.

[93] J. Nickolls and W. J. Dally. The gpu computing era. *IEEE Micro*, 30(2):56–69, 2010.

[94] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable parallel programming with cuda. *Queue*, 6(2):40–53, March 2008.

[95] C. Norbeg. Fluctuating lift on a circular cylinder: review and new measurements. *J. Fluids Struct*, 17(1):57 – 96, 2003.

[96] nVidia Corp. Just the facts. *Nvidia. Retrieved*, 2013.

[97] Christian Obrecht, Frédéric Kuznik, Bernard Tourancheau, and Jean-Jacques Roux. Scalable lattice boltzmann solvers for {CUDA} {GPU} clusters. *Parallel Computing*, 39(6–7):259 – 270, 2013.

[98] Kunle Olukotun. *Chip Multiprocessor Architecture: Techniques to Improve Throughput and Latency*. Synthesis Lectures on Computer Architecture. Morgan and Claypool Publishers, 1st edition, 2007.

[99] L. Ong and J. Wallace. The velocity field of a turbulent very near wake of a circular cylinder. *Exp. Fluids*, 20(6):441 – 453, 1996.

[100] E. Gross P. Bhatnagar and M. Krook. A model for collision processes in gases. i: small amplitude processes in charged and neutral one-component system. *Phys. Rev. E*, 94:511–525, 1954.

[101] Complex Physics Palabos, CFD. http://www.palabos.org/.

[102] C. S. Peskin. The immersed boundary method. *Acta Numerica*, 11:479 – 517, 2002.

[103] C.S. Peskin and B.F. Printz. Improved volume conservation in the computation of flows with immersed elastic boundaries. *J. Comput. Phys.*, 105:33, 1993.

[104] T. Pohl, M. Kowarchik, J. Wilke, and U. Rüde K. Iglberger. Optimization and profiling of the cache performance of parallel lattice boltzmann codes. *Parallel Processing Letters*, 13(4):549–560, 2003.

[105] Mont-Blanc Project. http://www.montblanc-project.eu/.

[106] GPGPU. General purpose computation using graphics hardware. http://www.gpgpu.org.

[107] Y. H. Qian, D. D'Humières, and P. Lallemand. Lattice bgk models for navier-stokes equation. *EPL (Europhysics Letters)*, 17(6):479, 1992.

[108] T. Ramirez, A. Pajuelo, O.J. Santana, and M. Valero. Runahead threads to improve smt performance. In *High Performance Computer Architecture, 2008. HPCA 2008. IEEE 14th International Symposium on*, pages 149–158, Feb 2008.

[109] S. Rapaka, T. Mansi, B. Georgescu, M. Pop, G.A. Wright, A. Kamen, and Dorin Comaniciu. Lbm-ep: Lattice-boltzmann method for fast cardiac electrophysiology simulation from 3d images. In Nicholas Ayache, Hervé Delingette, Polina Golland, and Kensaku Mori, editors, *Medical Image Computing and Computer-Assisted Intervention – MICCAI 2012*, volume 7511 of *Lecture Notes in Computer Science*, pages 33–40. Springer Berlin Heidelberg, 2012.

[110] P.R. Rinaldi, E.A. Dari, M.J. Vénere, and A. Clausse. A lattice-boltzmann solver for 3d fluid simulation on {GPU}. *Simulation Modelling Practice and Theory*, 25(0):163 – 171, 2012.

[111] A. Roth and G.S. Sohi. Speculative data-driven multithreading. In *High-Performance Computer Architecture, 2001. HPCA. The Seventh International Symposium on*, pages 37–48, 2001.

[112] D. Russell and Z. J. Wang. A cartesian grid method for modelling multiple moving objects in 2d incompressible viscous flows. *Journal of Computational Physics*, 191:177 – 205, 2003.

[113] Richard M. Russell. The cray-1 computer system. *Communications of ACM*, 21(1):63–72, January 1978.

[114] Anush Krishnana S. K. Layton and L. A. Barbaa. cuibm - a gpu-accelerated immersed boundary method. In 23^{rd} *International Conference on Parallel Computational Fluid Dynamics (ParCFD)*, 2011.

[115] Y. Saad. *Iterative Methods for Sparse Linear Systems*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2nd edition, 2003.

[116] N. Sakharnykh. Efficient tridiagonal solvers for adi methods and fluid simulation. In *NVIDIA GPU Technology Conference*, September 2010.

[117] Daniel Sanchez, George Michelogiannakis, and Christos Kozyrakis. An analysis of on-chip interconnection networks for large-scale chip multiprocessors. *ACM Transactions on Architecture and Code Optimization*, 7(1):4:1–4:28, May 2010.

[118] Jason Sanders and Edward Kandrot. *CUDA by Example: An Introduction to General-Purpose GPU Programming*. Addison-Wesley Professional, 2010.

[119] M. Schönherr, K. Kucher, M. Geier, M. Stiebler, S. Freudiger, and M. Krafczyk. Multi-thread implementations of the lattice boltzmann method on non-uniform grids for CPUs and GPUs. *Computers & Mathematics with Applications*, 61(12):3730 – 3743, 2011. Mesoscopic Methods for Engineering and Science - Proceedings of ICMMES-09 Mesoscopic Methods for Engineering and Science.

[120] Steven L. Scott. Synchronization and communication in the t3e multiprocessor. In Bill Dally and Susan J. Eggers, editors, *ASPLOS*, pages 26–36. ACM Press, 1996.

[121] C. de Boor S.D. Conte. *Elementary Numerical Analysis*, volume 1. McGraw-Hill, 1976.

[122] Sudip K. Seal. An accelerated recursive doubling algorithm for block tridiagonal systems. In *Proceedings of the 2014 IEEE 28th International Parallel and Distributed Processing Symposium*, IPDPS '14, pages 1019–1028, Washington, DC, USA, 2014. IEEE Computer Society.

[123] Sudip K. Seal, Kalyan S. Perumalla, and Steven P. Hirshman. Revisiting parallel cyclic reduction and parallel prefix-based algorithms for block tridiagonal systems of equations. *Journal of Parallel and Distributed Computing*, 73(2):273 – 280, 2013.

[124] Larry Seiler, Doug Carmean, Eric Sprangle, Tom Forsyth, Michael Abrash, Pradeep Dubey, Stephen Junkins, Adam Lake, Jeremy Sugerman, Robert Cavin, Roger Espasa, Ed Grochowski, Toni Juan, and Pat Hanrahan. Larrabee: A many-core x86 architecture for visual computing. In *ACM SIGGRAPH 2008 Papers*, SIGGRAPH '08, pages 18:1–18:15, New York, NY, USA, 2008. ACM.

[125] A. G. Shet, K. Siddharth, S. H. Sorathiya, A. M. Deshpande, S. D. Sherlekar, B. Kaul, and S. Ansumali. On Vectorization for Lattice Based Simulations. *International Journal of Modern Physics C*, 24:40011, December 2013.

[126] Aniruddha G. Shet, Shahajhan H. Sorathiya, Siddharth Krithivasan, Anand M. Deshpande, Bharat Kaul, Sunil D. Sherlekar, and Santosh Ansumali. Data structure and movement for lattice-based simulations. *Phys. Rev. E*, 88:013314, Jul 2013.

[127] M. Själander, M. Martonosi, and S. Kaxiras. *Power-Efficient Computer Architectures: Recent Advances*. Synthesis Lectures on Computer Architecture. Morgan and Claypool Publishers, December 2014.

[128] B. J. Smith. Architecture and applications of the hep multiprocessor computer system. *Proceesing of the International Society for Optical Engineering 298*, pages 241–248, 1981.

[129] Barry Smith. Petsc (portable, extensible toolkit for scientific computation). In David A. Padua, editor, *Encyclopedia of Parallel Computing*, pages 1530–1539. Springer, 2011.

[130] Ryan Smith. Intel's knights landing xeon phi coprocessor detailed, June 2014.

[131] Y. Sone. *Kinetic theory and fluid dynamics*. Birkhäuser, 2002.

[132] Thomas Sterling, Donald J. Becker, Daniel Savarese, John E. Dorband, Udaya A. Ranawake, and Charles V. Packer. Beowulf: A parallel workstation for scientific computation. In *In Proceedings of the 24th International Conference on Parallel Processing*, pages 11–14. CRC Press, 1995.

[133] Christopher P. Stone, Earl P. N. Duque, Yao Zhang, David Car, John D. Owens, and Roger L. Davis. GPGPU parallel algorithms for structured-grid CFD codes. In *Proceedings of the 20th AIAA Computational Fluid Dynamics Conference*, number 2011-3221, June 2011.

[134] H. S. Stone. An efficient parallel algorithm for the solution of a tridiagonal linear system of equations. *Journal of the ACM*, 20:27–38, 1973.

[135] Sauro Succi. *The Lattice Boltzmann Equation for Fluid Dynamics and Beyond (Numerical Mathematics and Scientific Computation)*. Numerical mathematics and scientific computation. Oxford University Press, USA, August 2001.

[136] P. N. Swarztrauber. A direct method for the discrete solution of separable elliptic equations. *SIAM J. Numer. Anal.*, 11:1136–1150, 1974.

[137] P. N. Swarztrauber. A parallel algorithm for solving general tridiagonal equations. 33:185–199, 1979.

[138] P. N. Swarztrauber and R. A. Sweet. The direct solution of the discrete Poisson equation on a disk. 10:900–907, 1973.

[139] Paul N. Swarztrauber and Roland A. Sweet. Vector and parallel methods for the direct solution of poisson's equation. *Journal of Computational and Applied Mathematics*, 27(1–2):241 – 263, 1989. Special Issue on Parallel Algorithms for Numerical Linear Algebra.

[140] R. A. Sweet. A generalized cyclic reduction algorithm. 11:506–520, 1974.

[141] R. A. Sweet. A cyclic reduction algorithm for solving block tridiagonal systems of arbitrary dimension. 14:706–720, 1977.

[142] K. Taira and T. Colonius. The immersed boundary method: A projection approach. *Journal of Computational Physics*, 225(2):2118 – 2137, 2007.

[143] Andrew V. Terekhov. A fast parallel algorithm for solving block-tridiagonal systems of linear equations including the domain decomposition method. *Parallel Computing*, 39(6–7):245 – 258, 2013.

[144] TOP500.org. TOP500 List June 2015.

[145] Ulrich Trottenberg, Cornelius W. Oosterlee, and Anton Schuller. *Multigrid*, volume 1. Academic Press, 2000.

[146] D.M. Tullsen, S.J. Eggers, and H.M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *Computer Architecture, 1995. Proceedings., 22nd Annual International Symposium on*, pages 392–403, June 1995.

[147] M. Tutar and A. E. Holdø. Computational modelling of flow around a circular cylinder in sub-critical flow regime with various turbulence models. *International Journal for Numerical Methods in Fluids*, 35(7):763–784, 2001.

[148] M. Uhlmann. An immersed boundary method with direct forcing for the simulation of particulate flows. *Journal of Computational Physics*, 209(2):448 – 476, 2005.

[149] Theo Ungerer, Borut Robič, and Jurij Šilc. A survey of processors with explicit multithreading. *ACM Comput. Surv.*, 35(1):29–63, March 2003.

[150] Andras Vajda. *Programming Many-Core Chips*. Springer, 2011.

[151] Pedro Valero-Lara. Parallel approaches for immersed-boundary method (solid-fluid interaction). *14th International Conference Computational and Mathematical Methods in Science and Engineering*, 4:1251–1262, 2014.

[152] Pedro Valero-Lara, Alfredo Pinelli, Julien Favier, and Manuel Prieto Matias. Block tridiagonal solvers on heterogeneous architectures. In *Proceedings of the 2012 IEEE 10th International Symposium on Parallel and Distributed Processing with Applications*, ISPA '12, pages 609–616, Washington, DC, USA, 2012. IEEE Computer Society.

[153] Pedro Valero-Lara, Alfredo Pinelli, and Manuel Prieto-Matias. Accelerating solid-fluid interaction using lattice-boltzmann and immersed boundary coupled simulations on heterogeneous platforms. *Procedia Computer Science*, 29(0):50 – 61, 2014. 2014 International Conference on Computational Science.

[154] Pedro Valero-Lara, Alfredo Pinelli, and Manuel Prieto-Matias. Fast finite difference poisson solvers on heterogeneous architectures. *Computer Physics Communications*, 185(4):1265 – 1272, 2014.

[155] Jeremiah van Oosten. Introduction to cuda 5.0. *nVidia*, 2014.

[156] Sven Verdoolaege, Juan Carlos Juega, Albert Cohen, José Ignacio Gómez, Christian Tenllado, and Francky Catthoor. Polyhedral parallel code generation for cuda. *ACM Trans. Archit. Code Optim.*, 9(4):54:1–54:23, January 2013.

[157] H. H. Wang. A parallel method for tridiagonal equations. *ACM Trans. Math. Softw.*, 7(2):170–183, June 1981.

[158] G. Wellein, T. Zeiser, G. Hager, and S. Donath. On the single processor performance of simple lattice boltzmann kernels. *Computers & Fluids*, 35(8–9):910 – 919, 2006. Proceedings of the First International Conference for Mesoscopic Methods in Engineering and Science.

[159] John F. Wendt and John David Anderson. *Computational Fluid Dynamics: An Introduction*. Springer, 2008.

[160] Markus Wittmann, Thomas Zeiser, Georg Hager, and Gerhard Wellein. Comparison of different propagation steps for lattice boltzmann methods. *Computers & Mathematics with Applications*, 65(6):924 – 935, 2013. Mesoscopic Methods in Engineering and Science.

[161] Michael Wolfe. Understanding the cuda data parallel threading model: A primer, December 2012.

[162] J. Wu and C.K. Aidun. Simulating 3d deformable particle suspensions using lattice boltzmann method with discrete external boundary force. *Int. J. Numer. Meth. Fluids*, 62:765–783, 2010.

[163] Ziheng Wu, Zhiliang Xu, Oleg Kim, and Mark Alber. Three-dimensional multi-scale model of deformable platelets adhesion to vessel wall in blood flow. *Philosophical Transactions of the Royal Society of London A: Mathematical, Physical and Engineering Sciences*, 372(2021), 2014.

[164] Next Generation of CFD XFlow. http://www.xflowcfd.com/.

[165] S. Xu and Z. J. Wang. An immersed interface method for simulating the interaction of a fluid with movies boundaries. *Journal of Computational Physics*, 216(2):454–493, 2006.

[166] Yao Zhang, Jonathan Cohen, and John D. Owens. Fast tridiagonal solvers on the gpu. *SIGPLAN Not.*, 45(5):127–136, January 2010.

[167] Hao Zhou, Guiyuan Mo, Feng Wu, Jiapei Zhao, Miao Rui, and Kefa Cen. {GPU} implementation of lattice boltzmann method for flows with curved boundaries. *Computer Methods in Applied Mechanics and Engineering*, 225–228(0):65 – 73, 2012.

[168] L. Zhu and C. S. Peskin. Interaction of two flapping filament in a flow soap film. *Physics of fluids*, 15:1954 – 1960, 2000.

[169] L. Zhu and C. S. Peskin. Simulation of a flapping flexible filament in a flowing soap film by the immersed boundary method. *Physics of fluids*, 179:452 – 468, 2002.

[170] Oleg Zikanov. *Essential Computational Fluid Dynamics*. John Wiley & Sons, Inc., 2010.